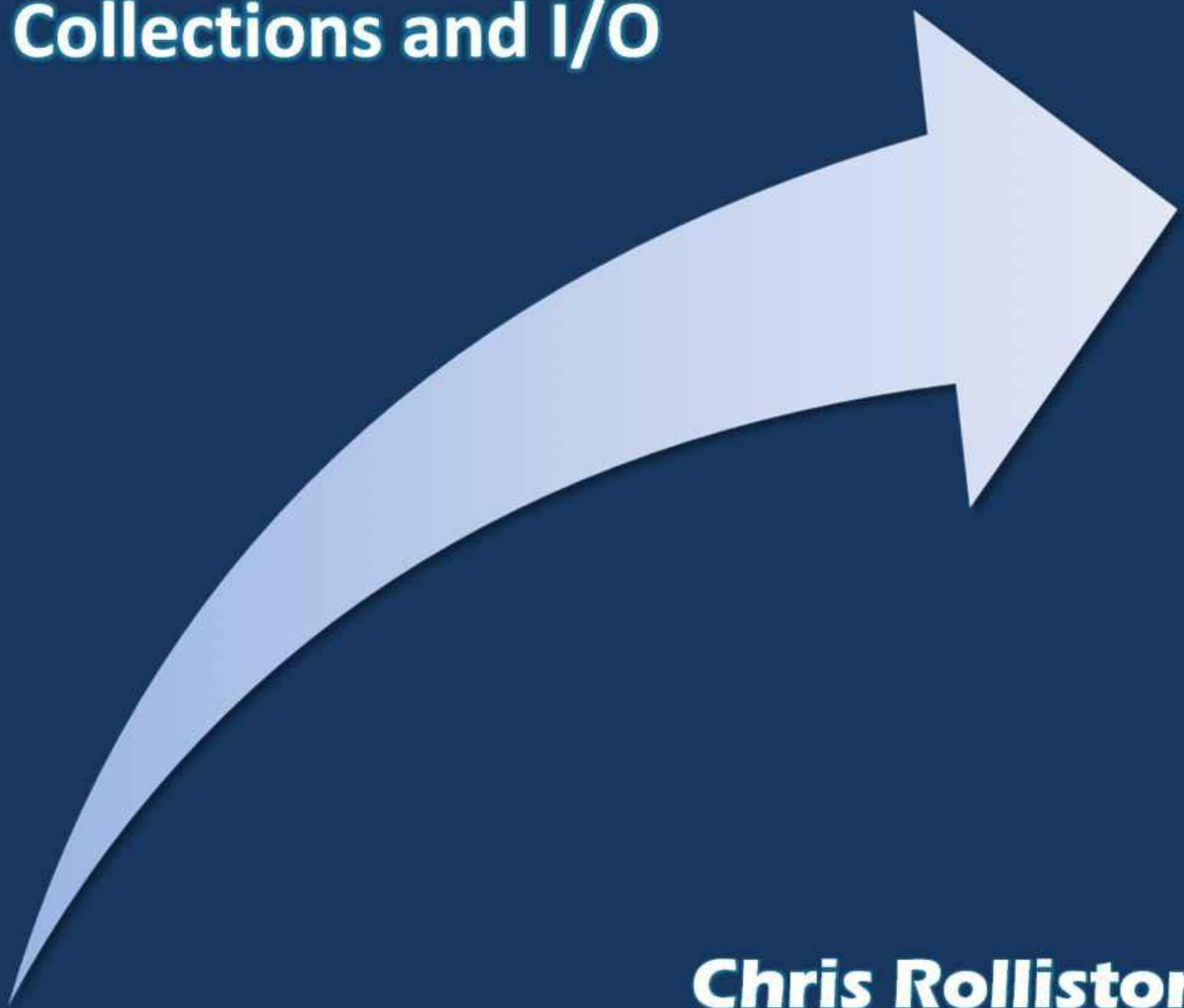
Delphi XE2 Foundations

Part 2: Strings, Arrays, Collections and I/O



Chris Rolliston

Delphi XE2 Foundations - Part 2

Updated 30 June 2012

Table of contents

General Introduction

Delphi is a complete software development environment for Windows, having its own language, libraries and 'RAD Studio' IDE. Now in its XE2 incarnation, Delphi continues to evolve. Most notably, XE2 supports writing applications for Mac OS X, an ability added in the context of numerous other extensions and improvements across recent releases.

Centred upon XE2, this book aims to provide a comprehensive introduction to the fundamentals of Delphi programming as they stand today. A contemporary focus means it has been written on the assumption you are actually using the newest version. Since a traditional strength of Delphi is how new releases rarely require old code to be rewritten, much of the text will apply to older versions too. However, the book does not explicitly highlight what will and what will not.

Conversely, a focus on fundamentals means a subject-matter of the Delphi language and wider runtime library (RTL). Topics such as the RAD Studio IDE and Visual Component Library (VCL — Delphi's longstanding graphical user interface framework) are therefore covered only incidentally. Instead, coverage spans from basic language syntax — the way Delphi code is structured and written in other words — to the RTL's support for writing multithreaded code, in which a program performs multiple operations at the same time.

If you are completely new to programming, this book is unlikely to be useful on its own. Nonetheless, it assumes little or no knowledge of Delphi specifically: taking an integrated approach, it tackles features that are new and not so new, basic and not so basic. In the process, it will describe in detail the nuts and bolts useful for almost any application you may come to write in Delphi.

About the Kindle edition of Delphi XE2 Foundations

Delphi XE2 Foundations is available in both printed and eBook versions. In eBook form, it is split into three parts; the part you are reading now is **part two**. All parts share the same 'general introduction', but beyond that, their content differs.

Chapter overview: part one

The first chapter of both the complete book and part one of the eBook set gives an account of the basics of the Delphi language. This details the structure of a program's source code, an overview of the typing system, and the syntax for core language elements. The later parts of the chapter also discuss the mechanics of error handling and the syntax for using pointers, a relatively advanced technique but one that still has its place if used appropriately.

Chapter two turns to consider simple types in more detail, covering numbers, enumerations and sets, along with how dates and times are handled in Delphi.

The third and fourth chapters look at classes and records, or in other words, Delphi's support for object-oriented programming (OOP). Chapter three considers the basics, before chapter four moves on to slightly more advanced topics such as metaclasses and operator overloading.

Chapter overview: part two

The first chapter of part two — chapter five of the complete book — considers string handling in Delphi. It begins with the semantics of the `string` type itself, before providing a reference for the RTL's string manipulation functionality, including Delphi's regular expressions (regex) support.

Chapter six discusses arrays, collections and custom enumerators, providing a reference for the first and second, and some worked-through examples of the third.

Chapters seven to nine look at I/O, focussing initially upon the basics of writing a console program in Delphi, before turning to the syntax for manipulating the file system (chapter seven), an in-depth discussion of streams (chapter eight), and Delphi's support for common storage formats (chapter nine).

Chapter overview: part three

Chapter ten — the first chapter of part three — introduces packages, a Delphi-specific form of DLLs (Windows) or dylibs (OS X) that provide a convenient way of modularising a monolithic executable. The bulk of this chapter works through a FireMonkey example that cross compiles between Windows and OS X.

Chapters eleven and twelve look at Delphi's support for dynamic typing and reflection — 'runtime-type information' (RTTI) in the language of Delphi itself. The RTTI section is aimed primarily as a reference, however example usage scenarios are discussed.

Chapter thirteen looks at how to interact with the native application programming interfaces (APIs). A particular focus is given to using the Delphi to Objective-C bridge for programming to OS X's 'Cocoa' API.

Chapter fourteen looks at the basics of writing and using custom dynamic libraries (DLLs on Windows, dylibs on OS X), focussing on knobbly issues like how to exchange string data in a language independent fashion. Examples are given of interacting with both Visual Basic for Applications (VBA) and C#/.NET clients.

Finally, chapter fifteen discusses multithreaded programming in Delphi, with a particular focus on providing a reference for the threading primitives provided by the RTL. This is then put to use in the final section of the book, which works through a 'futures' implementation based on a custom thread pool.

Trying out the code snippets

Unless otherwise indicated, the code snippets in this book will assume a console rather than GUI context. This is to focus upon the thing being demonstrated, avoiding the inevitable paraphernalia had by GUI-based demos.

To try any code snippet out, you will therefore need to create a console application in the IDE. If no other project is open, this can be done via the Tool Palette to the bottom right of the screen. Alternatively, you can use the main menu bar: select `File|New|Other...`, and choose `Console Application` from the `Delphi Projects` node. This will then generate a project file with contents looking like the following:

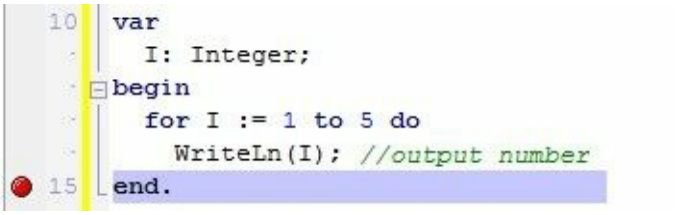
```
program Project1;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

Unless the context indicates otherwise, the `uses` to the final `end` should be overwritten with the code snippet.

By default, the console window will close when a debugged application finishes, preventing you from inspecting any output first. To avoid this, you can amend the final part of the code added to look like the following:

```
  //code...
  Write('Press ENTER to exit...',
  ReadLn;
end.
```

Alternatively, you can add a breakpoint at the end of the source code. Do this by clicking in the leftmost part of gutter beside the `end.`:

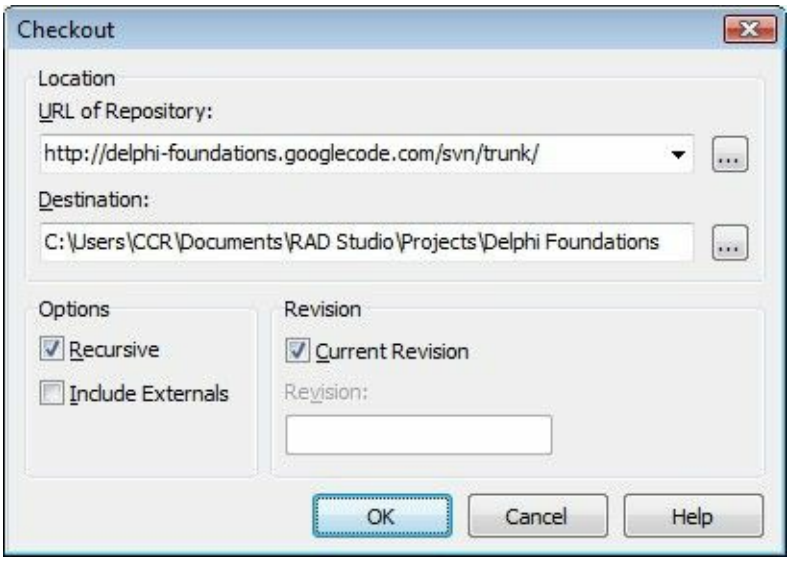


This will cause the IDE to regain control just before the program closes, allowing you to switch back to the console window to view the output. To allow the program to then close normally, switch back to the IDE and press either the 'Run' toolbar button or the F9 key, as you would have done to start the program in the first place.

Sample projects

Code for the more substantive examples in the book can be found in a Subversion repository hosted on Google Code. An easy way to download it is to use the RAD Studio IDE:

1. Choose `File|Open From Version Control...` from the main menu bar.
2. Enter `http://delphi-foundations.googlecode.com/svn/trunk/` for the URL, and a local folder of your choosing for the destination.



3. On clicking OK, a new dialog should come up listing the sample projects as they get downloaded.
4. Once complete, a third dialog will display allowing you to pick a project or project group to open. It isn't crucial to pick the right one though, since you can open any of the projects as normal afterwards.

Acknowledgements

I wish to thank the following, in no particular order: John Toelken for looking over chapters 1 and 3; Kenneth Cochran and Mark Walker for spotting some typos in chapters 4 and 8 respectively; Warren Postma for some concrete suggestions concerning chapter 13; Primož Gabrijelcic for technical editing of chapter 15; and Jasper Bongertz and James Chandler for assisting with tests on physical Kindle devices.

Chris Rolliston, 14 June 2012

5. String handling

This chapter builds on the basic account of strings in chapter 1, part 1 by considering Delphi's string support in depth. Beginning by looking in more detail at the `string` and `Char` types, the chapter will then turn to provide a detailed guide to the string handling functionality offered by the RTL. This will form the main bulk of the chapter. Following it, two relatively advanced topics (Unicode and C-style 'strings') will complete the discussion.

The string type

The `string` type in Delphi uses 'UTF-16' encoded characters ('Universal Transformation Format, 16 bit'), in which each element is two bytes in size. This is like strings in Java or C#, or indeed the old COM-based Visual Basic. Unlike strings in those other languages though, Delphi's are both mutable and have a 'copy on write' implementation. This makes Delphi strings very fast, since when you assign one string to another, merely a reference to the data of the first gets copied:

```
var
  S1, S2, S3: string;
begin
  S1 := 'Test';
  S2 := S1; //copies reference only
  S3 := S1; //copies reference only
```

If the first string is subsequently changed, the old data remains shared between the other strings:

```
  S1 := 'Different'; //now only S2 and S3 share data
```

Since the mechanics of mutability and copy-on-write are handled for you automatically, they are not things you need think about very often. In fact, their existence allows you to think *less* about the implementation of strings than otherwise. Thus, for many other languages, building a string by naïvely concatenating sub-strings in a repetitive fashion would be bad style. In Delphi, in contrast, it is idiomatic — the normal thing to do in other words — at least while the string being built up remains below megabytes in size:

```
function BuildString(List: TList<string>);
var
  I: Integer;
  S: string;
begin
  if List.Count = 0 then Exit('');
  Result := List[0];
  for I := 1 to List.Count - 1 do
    Result := Result + ', ' + List[I];
end;
```

A reference type that behaves like a value type

While `string` is a reference type, it frequently behaves as if it were a value type. Thus, the equality (=) and inequality (<>) operators always compare the data implicitly pointed to, rather than references to it. The fact strings are really pointers in disguise is witnessed by the fact you can cast a string to the `Pointer` type however:

```
var
  S1, S2: string;
begin
  { allocate a new string and assign second string to first }
  S1 := StringOfChar('W', 4);
  S2 := S1;
  { points to same data? }
  WriteLn(Pointer(S1) = Pointer(S2));     //output: TRUE
  { give the second string its own data }
  S2 := StringOfChar('W', 4);
  WriteLn(Pointer(S1) = Pointer(S2));     //output: FALSE
  WriteLn(S1 = S2);                       //output: TRUE
end.
```

The pretence of being a value type makes `string` different from dynamic arrays, which are otherwise conceptually similar:

```
var
  B1, B2: TBytes;
begin
  B1 := BytesOf('Test');
  B2 := BytesOf('Test');
  WriteLn(B1 = B2);      //output: FALSE
end.
```

String indexing

Strings can be indexed as if they were arrays of `Char`, in which the index is an integer that begins at 1 (*not* 0) for the first character. Individual elements can be both read and written to using string indexing:

```
var
  I: Integer;
  S: string;
```

```
begin
  S := 'hello world';
  for I := 1 to Length(S) do
  begin
    Write(S[I] + ' ');     //output: h e l l o
    S[I] := UpCase(S[I]);  //change the character
  end;
  WriteLn(S);              //output: HELLO WORLD
end.
```

Individual `Char` values can also be enumerated using a for/in loop, though this provides only read access:

```
procedure TestForIn;
var
  C: Char;
  S: string;
begin
  S := 'Hello world';
  for C in S do
    WriteLn(C);
end;
```

Returning length information (Length, ByteLength)

To return the number of `Char` values held in a string, call `Length`. Since this function is both inlined and returns a value stored in the string's implicit header, it is a very fast. `System.SysUtils` also provides a `ByteLength` function that returns the number of bytes taken up by a string's data, which will be `Length` multiplied by `SizeOf(Char)`, i.e. 2.

Other string types

Up until now, I have been talking about 'the' string type in Delphi. In reality, a number of string types exist:

- `UnicodeString` is what `string` currently maps to. Its characters are UTF-16 encoded, with its instances being reference counted, having copy on write semantics, and supporting 1-based indexing in which each element is a `WideChar`/`Char` value.
- `WideString` wraps the COM `BSTR` type on Windows (it just maps to `UnicodeString` on OS X). Like `UnicodeString`, it uses UTF-16 and supports 1-based indexing in which each element is a `WideChar`/`Char` value. However, it uses the COM memory allocator under the bonnet, and is neither reference counted nor has copy on write semantics. In practical terms this makes it slower than `UnicodeString`/`string`.
- `AnsiString` defines strings that are encoded using the current Windows 'Ansi' codepage. Its instances are reference counted, with copy-on-write semantics and supporting 1-based indexing. However, the elements of an `AnsiString` are typed to `AnsiChar` not `WideChar`. For many versions `string` was mapped to this type.
- `AnsiString(n)`, where n is a `Word`/`UInt16` number, defines an `AnsiString` type (1-indexed, reference counted, etc.) for the specified 'Ansi' codepage. Individual types must be predeclared before use:

```
type
  CyrillicAnsiString = type AnsiString(1251);
```

- `UTF8String` is equivalent to `AnsiString(CP_UTF8)`.
- `RawByteString` should be used for the parameter type of a routine that accepts any sort of 'Ansi' string. If simply `AnsiString` (or `UTF8String`) is used, character data will be converted whenever an 'Ansi' string of a different code page is passed.
- `string[n]`, where n is a number between 1 and 255, defines a fixed size 'Ansi' string.
- `ShortString` is equivalent to `string[255]`; this was the native string type in Delphi's very first version.

In general, the only string type you need care about is `string`. This is because it is the primary type used by Delphi's standard libraries, and as such, will have to be used at various points even if you employ another sort of string internally.

Beyond it, `WideString` is used when working with Microsoft's Component Object Model (COM), and `UTF8String` when converting between Delphi's native UTF-16 encoding and UTF-8. That these conversions may be needed is because UTF-8 tends to be used for text that is stored or persisted, notwithstanding the fact UTF-16 is a popular in-memory format (aside from Delphi, the Windows API, Microsoft .NET, Java, and Apple's Cocoa libraries on OS X have all standardised on it).

Conversions between `string` and `UTF8String` are lossless since they are both types of 'Unicode' encoding (`UnicodeString`

should really be called `UTF16String`!). In simple terms, an 'encoding' is a defined pattern for mapping human-readable characters to the numbers that computers think in, and 'Unicode' is the modern set of standards for this. In contrast to older encodings, Unicode aims to provide a set of numbers that covers all characters in all scripts known to humankind, or thereabouts; older encodings, in contrast, would cover only one or two scripts, and in mutually exclusive ways.

Conversions between different string types is automatic — just assign an instance of one type to an instance of another — however the compiler will usually issue a warning when you do that, either to confirm that a conversion will be happening, or that moreover, the conversion may be lossy. To get rid of the warning, use the typecast syntax — `NewStr := NewStrType(OldStr)`:

```
var
  UTF8: UTF8String;
  UTF16: string;
begin
  { As UTF-8 and UTF-16 are both Unicode encodings, converting
    between them will not corrupt the data }
  UTF8 := 'These are some chess pieces: ♔♕♖♗♘';
  ShowMessage('Number of UTF-8 elements ' +
    '(1 byte per element): ' + IntToStr(Length(UTF8)));
  UTF16 := string(UTF8);
  ShowMessage(UTF16);
  ShowMessage('Number of UTF-16 elements ' +
    '(2 bytes per element): ' + IntToStr(Length(UTF16)));
  UTF8 := UTF8String(UTF16);
  ShowMessage('Number of elements back in UTF-8: ' +
    IntToStr(Length(UTF8)));
```

Legacy string types

Of the other string types, `string[x]` and `ShortString` are for backwards compatibility with very old code — you shouldn't use them for greenfield development. This leaves the 'Ansi' string types. These relate to older versions of Windows, in particular Windows 3.x, 9x and ME. Not natively supporting Unicode, those operating systems worked with different 'code pages' for different scripts instead — English, French and German shared one code page ('Western European'), Greek had another, Cyrillic a fourth, and so on, with each code page being assigned its own number:

- 1250 for the 'Central European' code page
- 1251 for Cyrillic
- 1252 for 'Western European' (alias 'Latin 1')
- 1253 for Greek
- 1254 for Turkish
- 1255 for Hebrew
- 1256 for Arabic
- 1257 for Baltic languages
- 1258 for Vietnamese (this is very similar but not identical to Latin 1)
- 874 for Thai
- 932 for Japanese Shift-JIS
- 936 for Simplified Chinese GBK
- 949 for Korean
- 950 for Traditional Chinese Big5

Collectively, these became referred to as 'Ansi' code pages, due to the fact Microsoft based the 'Western European' one on a draft American National Standards Institute (ANSI) standard. The relation to the Institute is no stronger than that however — none of the Microsoft code pages are actually finalised ANSI standards.

Back in Delphi, the `AnsiString` syntax allows defining string types whose `AnsiChar` elements (*not* `WideChar`/`Char`) are encoded using a specific 'Ansi' code page. Similar to `UTF8String`, this makes converting to and from native UTF-16 strings very easy, assuming you know beforehand what legacy code page to use. Normally this means the 'ACP' (Ansi Code Page) set at the operating system level, in which case you should use the non-parameterised version of `AnsiString`:

```
uses System.SysUtils, Vcl.Dialogs;
```

```
var
  LegacyStr: AnsiString;
  NativeStr: string;
begin
  LegacyStr := 'What a hullabaloo encodings are!';
  NativeStr := string(LegacyStr);
  ShowMessage(NativeStr);           //shows correctly
  NativeStr := 'This ↑↓ will probably be a lossy conversion';
  ShowMessage(NativeStr);           //shows correctly
  LegacyStr := AnsiString(NativeStr);
  ShowMessage(string(LegacyStr));   //roundtrip corrupted data
end.
```

When a specific code page needs to be used, predeclare a parameterised `AnsiString` for it:

```
type
  GreekAnsiString = type AnsiString(1253);

var
  GreekAnsi: GreekAnsiString;
  NativeStr: string;
begin
  NativeStr := 'γειά σου κόσμε';
  ShowMessage('Number of native string elements ' +
    '(2 bytes/element): ' + IntToStr(Length(NativeStr)));
  //convert to the Greek 'Ansi' format
  GreekAnsi := GreekAnsiString(NativeStr);
  ShowMessage('Number of Greek "Ansi" string elements ' +
    '(1 byte/element): ' + IntToStr(Length(GreekAnsi)));
  //convert back to native UTF-16
  NativeStr := string(GreekAnsi);
  ShowMessage(NativeStr);
```

Since it was convenient to implement it that way, `UTF8String` is technically an 'Ansi' string too in Delphi terms, however this shouldn't confuse you — UTF-8 isn't itself an 'Ansi' code page.

Character types

Delphi supports two 'fundamental' character types, `AnsiChar` and `WideChar`, and one 'generic' one, `Char`, that currently maps to `WideChar`, and will continue to do so in the foreseeable future. Where a `string` is conceptually a sequence of `Char` values, so an `AnsiString` (or `UTF8String`) is conceptually a sequence of `AnsiChar` values. The following therefore outputs `Hiya world`:

```
var
  Elem: AnsiChar;
  UTF8: UTF8String;
begin
  UTF8 := 'Hiya world';
  for Elem in UTF8 do
    Write(Elem);
```

Typically, one `Char`, `AnsiChar` or `WideChar` value is one human-readable character or punctuation mark, for example the letter `F` or a space character.

Showing its Pascal heritage, character values in Delphi are considered distinct from integers, unlike in C or C++. It is therefore illegal to assign an integer to a `Char` or vice versa without a typecast, or in the case of a `Char` to integer conversion, a call to the `Ord` standard function. Nonetheless, every `Char` or `WideChar` value has a well-defined ordinal value that adheres to the Unicode standard. For example, the minuscule Latin letter 'a' has an ordinal value of 97, and the euro sign (€) the ordinal value 8,364:

```
var
  SmallACh, EuroCh: Char;
begin
  SmallACh := 'a';
  EuroCh := '€';
  WriteLn(Ord(SmallACh)); //outputs 97
  WriteLn(Ord(EuroCh));   //outputs 8364
end.
```

The ordinal values for the single byte `AnsiChar` are the same within the ASCII range (i.e., 0 to 127 inclusive), but diverge after that, both with respect to the `Char`/`WideChar` values and between different code pages:

```
type
  CyrillicAnsiString = type AnsiString(1251);
  GreekAnsiString = type AnsiString(1253);
var
  Cyrillic: CyrillicAnsiString;
  Greek: GreekAnsiString;
begin
  Cyrillic := AnsiChar(198);
  Greek := AnsiChar(198);
  ShowMessage('AnsiChar(198) interpreted as Cyrillic Ansi: ' +
    string(Cyrillic));                    //outputs Ж
  ShowMessage('AnsiChar(198) interpreted as Greek Ansi: ' +
    string(Greek));                       //outputs Z
  ShowMessage('Char(198): ' + Char(198)); //outputs Æ
```

Literals and constants

A `Char` constant can be declared either using a literal (e.g. `'c'`) or an ordinal number. In the case of defining a constant with a number, you must prefix it with a hash symbol (#), otherwise it will be interpreted as an integer of some sort rather than a character and not compile. Also, for characters outside the ASCII range (i.e., with ordinal numbers greater than 127), it is advisable to use a four digit hexadecimal number. If you don't do this, the compiler may not assume you are using a Unicode ordinal:

```
const
  AcuteDiacritic = #$0301;  //using hexadecimal notation
  CapitalA       = 'A';     //character literal
  Tab            = #9;      //using decimal notation
  PeaceSign      = '☮';     //another literal
```

When typing non-ASCII characters directly into source code, you should ensure Unicode PAS files are being used — do this by right-clicking in the editor, selecting `File Format`, then `UTF8`.

Character constants can be combined to produce string constants. When doing this, using the + operator is optional so long as there are no spaces in between the individual constituent constants:

```
const
  UsualWay = 'Col 1' + #9 + 'Col 2' + #9 + 'Col 3';
```

```
  AnotherWay = 'Col 1'#9'Col 2'#9'Col 3';
```

In this example, both `UsualWay` and `AnotherWay` define the same string value.

Working with Char values: the TCharacter type

Declared in the `System.Character` unit, `TCharacter` is a class entirely composed of static methods that work on individual `Char` values. For example, its `ToUpper` and `ToLower` functions convert a character's casing:

```
uses System.Character, Vcl.Dialogs;

begin
  ShowMessage(TCharacter.ToUpper('a')); //outputs A
  ShowMessage(TCharacter.ToLower('Æ')); //outputs æ
end.
```

The majority of `TCharacter`'s methods concern finding out what type of character a given `Char` value is:

```
class function IsControl(C: Char): Boolean;
class function IsDigit(C: Char): Boolean;
class function IsHighSurrogate(C: Char): Boolean;
class function IsLetter(C: Char): Boolean;
class function IsLetterOrDigit(C: Char): Boolean;
class function IsLowSurrogate(C: Char): Boolean;
class function IsNumber(C: Char): Boolean;
class function IsPunctuation(C: Char): Boolean;
class function IsSeparator(C: Char): Boolean;
class function IsSurrogate(Surrogate: Char): Boolean;
class function IsSurrogatePair(HighSurrogate,
  LowSurrogate: Char): Boolean;
class function IsSymbol(C: Char): Boolean;
class function IsUpper(C: Char): Boolean;
class function IsWhiteSpace(C: Char): Boolean;
```

The meaning of a 'control' character, 'digit' and so forth is as per the Unicode standard. Consequently, control characters include the ASCII tab character (#9) but not the ASCII space character (#32), and 'numbers' include fraction characters like ½, which however *aren't* deemed 'symbols'.

For convenience, all methods of `TCharacter` have corresponding freestanding routines that allow you to drop the '`TCharacter`.' and use just the method name instead:

```
procedure Foo(const Ch: Char);
begin
  if TCharacter.IsDigit(Ch) then WriteLn('Is a digit');
  if IsDigit(Ch) then WriteLn('Is still a digit');
end;
```

Aside from the `IsXXX` methods, `GetUnicodeCategory` is also provided to act as a sort of 'master' function:

```
type
  TUnicodeCategory = (ucControl, ucFormat, ucUnassigned, ucPrivateUse, ucSurrogate, ucLowercaseLetter, ucModifierLetter,

class function GetUnicodeCategory(C: Char): TUnicodeCategory;
```

`TUnicodeCategory` is a 'scoped' enumeration, which means you must always fully qualify: `TUnicodeCategory.ucCurrencySymbol` not just `ucCurrencySymbol`.

When `IsNumber` returns `True`, `GetUnicodeCategory` will return one of `ucDecimalNumber`, `ucLetterNumber` or `ucOtherNumber`. 'Decimal' numbers include 1 and 4, 'letter' numbers include Roman numerals like Ⅳ, and 'other' numbers include fractions. When any of the three holds, call the `GetNumericValue` method to get a `Double` containing the number denoted by the character. For example, `GetNumericValue('Ⅳ')` will return 4, and `GetNumericValue('½')` will return 0.25. Only single Unicode characters are supported; neither the string `'IV'` (i.e., a capital 'I' followed by a 'V') nor `'1/2'` will work

String manipulation routines

As implemented by the RTL, the majority of Delphi's string handling functionality comes in the form of standalone routines declared across the `System`, `System.SysUtils` and `System.StrUtils` units. With a couple of exceptions, these are all of functions that return new strings, as opposed to procedures that change the source strings in place:

```
var
 S: string;
begin
  ReadLn(S);
  Trim(S);      //wrong!
  S := Trim(S); //right
end.
```

For historical reasons, `System.SysUtils` and `System.StrUtils` frequently employ a slightly odd naming convention, in which functions are 'doubled up' so that there are versions named with and without an 'Ansi' prefix — for example, alongside `SameText` stands `AnsiSameText`. This can be confusing even for experienced uses, since the `AnsiXXX` functions don't actually work with `AnsiString` types. Rather, the prefix means they definitely work with characters outside the ASCII range where the plain versions may not.

Creating repeating strings (StringOfChar, DupeString)

To create a string of a single character that repeats several times, use the `StringOfChar` function:

```
S := StringOfChar('z', 5);
```

This assigns `'zzzzz'` to `S`. Alternatively, to create a string that repeats another string several times, use `DupeString` from `System.StrUtils`:

```
S := 'He said:' + DupeString(' blah', 3);
```

The sets `S` to `'He said: blah blah blah'`.

Comparing complete strings strictly (=, <>, CompareXXX and AnsiCompareXXX)

When used with strings, the equality (=) and inequality (<>) operators look for four things in turn:

- Whether the two operands point to the same data
- Whether one or both strings are empty ('').
- Whether their respective lengths don't match.
- Each character's ordinal value in turn.

Strings also support the smaller-than (<), smaller-than-or-equals (<=), greater-than (>) and greater-than-or-equals (>=) operators. An empty string is considered smaller than a non-empty string, otherwise the ordinal values of individual characters working left to right are used: 'taste' is therefore smaller than 'test', despite being longer, because the letter 'a' has a lower ordinal value (97) than the letter 'e' (101).

If you wish to put a number to a string comparison test, call the `CompareStr` function of `System.SysUtils`. This returns 0 for an exact match, a negative number when the first string is less than the second, and a positive number when the second string is greater than the first. Calling `CompareStr` instead of using operators is particularly preferable when you want to do different things for each of the three possible answers, since it will involve comparing the strings only once rather than multiple times:

```
case CompareStr(S1, S2) of
  Low(Integer)..-1: { do something } ;
  0:                { do something else } ;
  1..High(Integer): { do something else again } ;
end;
```

Within the ASCII range, this sort of test will work well. It won't so much for strings with characters that aren't punctuation marks or unardorned 'English' letters though. If that is a concern, you should either pass `loUserLocale` as an extra parameter to `CompareStr`, or call `AnsiCompareStr` instead (both do the same thing):

```
var
  S1, S2: string;
begin
  S1 := 'weiß';
  S2 := 'weit';
  WriteLn(CompareStr(S1, S2));               //outputs 107
  WriteLn(CompareStr(S1, S2, loUserLocale)); //outputs -1
```

```
  WriteLn(AnsiCompareStr(S1, S2));          //outputs -1
end.n
```

Here, the eszett (i.e., the ß character) should sort before the letter 't' according to the rules of the German language. Being a non-ASCII character however, its ordinal value is greater (ASCII characters come first), leading to the simpler version of `CompareStr` returning the 'wrong' result. Use of `loUserLocale` or `AnsiCompareStr` fixes matters.

Both `CompareStr` and `AnsiCompareStr` are case sensitive, so that 'TEST' is not considered the same as 'test'. The case insensitive versions are `CompareText` and `AnsiCompareText`. Beyond case insensitivity, their semantics match those of their case sensitive equivalents.

Further variants are `SameText` and `AnsiSameText`, which are case insensitive like `CompareText` and `AnsiCompareText` but return a Boolean instead of an integer. Lastly, if you are comparing file names, `SameFileName` will handle for you differences between case sensitive and case insensitive file systems, along with idiosyncrasies in (for example) the way OS X stores file names.

Comparing complete strings approximately (ResemblesText, Soundex)

`System.StrUtils` declares a `ResemblesText` function that determines whether two words are 'similar'. By default, the 'Soundex' algorithm is used, which you can access directly via the `Soundex` function and related routines:

```
uses
  System.StrUtils;

var
  S1, S2: string;
  Similar: Boolean;
begin
  S1 := 'played';
  S2 := 'plaid';
  WriteLn('The Soundex code for "' + S1 +
          '" is ' + Soundex(S1));         //P430
  WriteLn('The Soundex code for "' + S2 +
          '" is ' + Soundex(S2));         //P430
  Similar := ResemblesText(S1, S2));      //TRUE
end.
```

If you wish, you can plug in your own algorithm by assigning the `ResemblesProc` procedural pointer to an appropriate function. If so, the custom implementation must take two string arguments and returns a `Boolean`. For example, the following alternative implementation returns `True` on either a strict case insensitive match, or when just an '-ing' suffix is the difference between the two strings:

```
uses
  System.SysUtils, System.StrUtils;

function MyResemblesTextImpl(const S1, S2: string): Boolean;

  function StripSuffix(const S: string): string;
  begin
    if EndsText('ing', S) then
      Result := Copy(S, 1, Length(S) - 3)
    else
      Result := S;
  end;
begin
  Result := SameText(StripSuffix(S1), StripSuffix(S2), loUserLocale);
end;
```

Here's it in action:

```
procedure Test;
begin
  if ResemblesText('shoot', 'SHOOTING') then
    WriteLn('Yes')
  else
    WriteLn('No');
  if ResemblesText('shot', 'shoot') then
    WriteLn('Yes')
  else
    WriteLn('No');
end;

begin
  Test; //uses default impl., i.e. Soundex; outputs No, Yes
```

```
  ResemblesProc := MyResemblesTextImpl;
  Test; //uses custom implementation; outputs Yes, No
end.
```

Pattern matching (MatchesMask, TMask)

Declared in `System.Masks`, the `MatchesMask` routine provides simple wildcard matching functionality. It works in a similar fashion to the `Like` operator of Visual Basic, albeit in the form of a function call:

```
if MatchesMask(MyStr, '*[a-z]') then
  WriteLn('Ends with an unadorned letter');
if MatchesMask(MyStr, '[0-9]*') then
  WriteLn('Starts with a digit');
if MatchesMask(MyStr, 'r*g') then
  WriteLn('Starts with an "r" and ends with a "g"');
if MatchesMask(MyStr, '????ed') then
  Writeln(MyStr, 'Has 6 characters ending in "ed"');
```

Comparisons are essentially case insensitive. The supported wildcards are * and ? — where an asterisk matches zero or more unspecified characters, a question mark matches a single unspecified character. Outside of square bracket pairs, any other character contained in the match string is one that *must* be matched if `MatchesMask` is to return `True`.

Square brackets themselves allow you to specify a set of acceptable characters. Within any square bracket pair, a range of characters can be specified using a hyphen, so that [a-d] matches for ‘a’, ‘b’, ‘c’ or ‘d’. To specify a non-continuous set, include all the characters concerned without any spaces: [a-ctwz] therefore matches ‘a’, ‘b’, ‘c’, ‘t’, ‘w’ or ‘z’.

The final piece of supported syntax allows specifying a set of characters that *cannot* appear at the position in question. For this, you define a square bracketed set as normal, only prefixing the character list with an exclamation mark. Thus, the following returns `True` when `MyStr` does not begin with the letters ‘a’, ‘s’ or ‘z’:

```
if MatchesMask(MyStr, '[!asz]*') then ...
```

Under the hood, `MatchesMask` works by constructing a temporary `TMask` object. If you will be testing multiple strings against the same mask, consider creating a `TMask` directly instead of calling `MatchesMask` multiple times:

```
function All4DigitStrings(const Arr: array of string): Boolean;
var
  Mask: TMask;
  S: string;
begin
  Result := False;
  Mask := TMask.Create('[0-9][0-9][0-9][0-9]');
  try
    for S in Arr do
      if not Mask.Matches(S) then Exit;
  finally
    Mask.Free;
  end;
  Result := True;
end;

//...

RetVal := All4DigitStrings(['7631', '1573', '1630']); //True
```

MatchesMask and TMask limitations

Whether you use either `MatchesMask` or `TMask` directly, two significant limitations exist: you can’t turn off case insensitivity for ‘ordinary’ letters, and parsing is somewhat ASCII-centric. The second issue means case insensitivity actually disappears if you specify characters outside of the ASCII range. Thus, where `MatchesMask('cafe', '*E')` returns `True`, `MatchesMask('café', '*É')` returns `False`. Nonetheless, match criteria do support any characters within the ‘Latin-1’ range (meaning, characters than have an ordinal number according to the Unicode standard of 255 or below). The workaround for accented letters not being treated case insensitively is therefore to specify both lower and upper case versions in the mask:

```
const
  Vowels = 'aeiouÀÁÂÃÄÅÆÈÉÊËÌÍÎÏÒÓÔÕÖØÙÚÛÜàáâãäåæèéêëìíîïðòóôõöùúûü';
begin
  if MatchesMask('café', '*[aeiou]') then
    Write('Got it first time around!')
  else
    Writeln('Eek! Failed first time around!');
  if MatchesMask('café', '*[' + Vowels + ']') then
```

```
    Write('Got it second time around!')
  else
    Writeln('Eek! Failed again!');
end.
```

In this example, the first test fails but the second succeeds.

Checking for sub-strings (StartsXXX, EndsXXX, ContainsXXX)

While you could use `MatchesMask` to determine whether one string (as opposed to one string pattern) starts another, there is less overhead if you call either `StartsStr` for a case-sensitive check or `StartsText` for a case-insensitive one. Both functions are declared in `System.StrUtils`, and both are matched with `Ansi-` versions; unlike with routines such as `CompareText`/`AnsiCompareText` though, both versions support Unicode equally.

Corresponding to `StartsXXX` are `EndsStr` and `EndsText`, which test for whether the contents of one string matches the end of another, and `ContainsStr` and `ContainsText`, which test for whether the contents of one string is *somewhere* within the contents of another. Watch out for how the argument order is reversed with `ContainsStr` and `ContainsText` though — with the other functions, the sub-string comes first and the main string second, whereas the `ContainsXXX` functions take the sub-string second and the main string first:

```
var
  S1, S2: string;
begin
  S1 := 'Brown bears fart VERY LOUDLY';
  S2 := 'Polar bears fart quite loudly';
  //next line produces False, False
  WriteLn('StartsStr with "brown":    ',
    StartsStr('brown', S1), ', ', StartsStr('brown', S2));
  //next line produces True, False
  WriteLn('StartsText with "brown":   ',
    StartsText('brown', S1), ', ', StartsText('brown', S2));
  //next line produces False, True
  WriteLn('EndsStr with "loudly":     ',
    EndsStr('loudly', S1), ', ', EndsStr('loudly', S2));
  //next line produces True, True
  WriteLn('EndsText with "loudly":    ',
    EndsText('loudly', S1), ', ', EndsText('loudly', S2));
  //next line produces True, True
  WriteLn('ContainsStr with "bears":  ',
    ContainsStr(S1, 'bears'), ', ', ContainsStr(S2, 'bears'));
  //next line produces True, True
  WriteLn('ContainsText with "BEARS": ',
    ContainsText(S1, 'BEARS'), ', ', ContainsText(S2, 'BEARS'));
end.
```

Locating sub-strings (Pos, PosEx, SearchBuf)

The `Pos` standard function returns the 1-based index of a sub-string in a string, or 0 if the sub-string can't be found. `PosEx` (declared in `System.StrUtils`) extends `Pos` to take an index to start searching from, rather than always just starting from the beginning:

```
var
  Index: Integer;
begin
  Index := Pos('é', 'Café Touché');
  WriteLn(Index);                                  //outputs 4
  Index := PosEx('é', 'Café Touché', Index + 1);
  WriteLn(Index);                                  //outputs 11
end.
```

Both `Pos` and `PosEx` work in terms of `Char` indexes.

A slightly more advanced way to search for a sub-string is with the `SearchBuf` function, which is declared in `System.StrUtils`. This takes a `PChar` buffer to search in rather than a `string`, and returns a pointer to the found sub-string within the buffer, or `nil` if it isn't found. Here's how it is declared:

```
type
  TStringSeachOption = (soDown, soMatchCase, soWholeWord);
  TStringSearchOptions = set of TStringSeachOption;

function SearchBuf(Buf: PChar; BufLen: Integer;
  SelStart, SelLength: Integer; SearchString: string;
  Options: TStringSearchOptions): PChar;
```

The `SelStart` parameter is zero indexed. If both it and `SelLength` is 0, and `soDown` is included in `Options`, then the whole buffer is searched. Conversely, if `SelStart` is 0 and `soDown` *isn't* included in `Options`, `nil` will be returned given there's nowhere to search prior to the start. Specifying `soMatchCase` turns off case insensitivity, and `soWholeWord` returns only whole word matches. A 'word' here is defined as any continuous sequence of alphanumeric characters. This means a hyphenated phrase like 'semi-final' is considered two separate words.

When searching downwards, the effective starting position is `SelStart + SelLength`; when searching upwards, `SelStart` alone. This deliberately matches the semantics of an edit or memo-type control, for which `SearchBuf` is ideally suited for implementing basic find functionality.

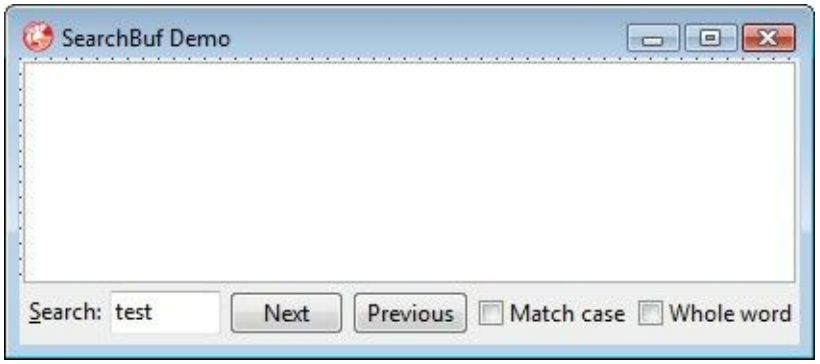
SearchBuf example

Putting `SearchBuf` to use, the following code implements a generalised find routine for VCL `TCustomEdit` descendants like `TEdit`, `TMemo` and `TRichEdit`:

```
uses
  System.SysUtils, System.StrUtils,
  Vcl.StdCtrls, Vcl.ComCtrls, Vcl.Dialogs;

procedure DoEditSearch(Control: TCustomEdit; const S: string;
  MatchCase, MatchWholeWord, Down: Boolean);
var
  Options: TStringSearchOptions;
  Text: string;
  StartPtr, EndPtr: PChar;
begin
  if S = '' then
  begin
    Beep;
    Exit;
  end;
  if Down then Options := [soDown] else Options := [];
  if MatchCase then Include(Options, soMatchCase);
  if MatchWholeWord then Include(Options, soWholeWord);
  Text := Control.Text;
  { Next line works around rich edit quirk (uses CR not the
    Windows standard CR + LF for line breaks) }
  if Control is TCustomRichEdit then
    Text := AdjustLineBreaks(Text, tlbsLF);
  StartPtr := PChar(Text);
  EndPtr := SearchBuf(StartPtr, Length(Text), Control.SelStart,
    Control.SelLength, S, Options);
  if EndPtr = nil then
  begin
    MessageDlg('"' + S + '" not found', mtInformation, [mbOK], 0);
    Exit;
  end;
  Control.SelStart := EndPtr - StartPtr;
  Control.SelLength := Length(S);
end;
```

Here, we first test for whether we have anything to search for; if we have, the options set is built up, the text to search in retrieved from the control, and `SearchBuf` called. If nothing is found (indicated by `SearchBuf` returning `nil`), a message box is shown stating the fact, otherwise the new selection is set (subtracting one `PChar` from another finds the number of `Char` values between them).

You can put `DoEditSearch` itself to use like this: on a new VCL form, add a `TMemo` or `TRichEdit` called `redEditor`, an edit control called `edtSearchStr` for the search box, buttons for 'Find Next' and 'Find Previous' commands, and check boxes for the 'Match case' and 'Match words only' options:



The Find Next button's `OnClick` event can then be handled like this:

```
DoEditSearch(redEditor, edtSearchStr.Text,
  chkMatchCase.Checked, chkMatchWholeWord.Checked, True);
```

Similarly, the `OnClick` event for the 'Find Previous' button can be handled as thus:

```
DoEditSearch(redEditor, edtSearchStr.Text,
  chkMatchCase.Checked, chkMatchWholeWord.Checked, False);
```

To finish things off, set the `TMemo` or `TRichEdit`'s `HideSelection` property to `False`. This ensures the coloured background for the currently-selected text doesn't disappear when the memo loses the keyboard focus.

Retrieving sub-strings (Copy, AnsiLeftStr, AnsiMidStr, AnsiRightStr)

The `Copy` standard function is the main way to get a sub-string of a string. It takes three parameters: the source string, the index to copy from (where index 1 is the first character), and the number of elements to copy. Conveniently, it is not an error to request more characters than can actually be returned; when you do that, a sub-string from the specified start position to the source string's end is returned:

```
var
  S1, S2: string;
begin
  S1 := 'This is not a very long string';
  S2 := Copy(S1, 9, 3);
  WriteLn(S2);              //outputs 'not'
  S2 := Copy(S1, 15, 999);
  WriteLn(S2);              //outputs 'very long string'
end.
```

As with most string functions, `Copy` works with `Char` indices and counts; if you have exotic strings that contain surrogate pairs, in which a single human-readable character takes up two `Char` values, `Copy` will therefore quite happily chop a pair into two. If that is a concern, you can use the higher-level `AnsiLeftStr`, `AnsiMidStr` and `AnsiRightStr` functions of the `StrUtils` unit. These count a surrogate pair as a single character, though at the cost of being a lot slower than naïve calls to `Copy`. `System.StrUtils` also has `LeftStr`, `MidStr` and `RightStr` functions, but since they are implemented merely as super-simple wrappers round `Copy`, you might as well use `Copy` directly if surrogates aren't a concern, which they normally won't be.

Deleting, inserting and replacing sub-strings (Delete, Insert, StuffString, StringReplace)

To delete a specified number of `Char` values from a specified position in a string, call the `Delete` procedure. This takes three parameters: the string to modify, the position or index to delete from, and the number of elements to delete:

```
var
  S: string;
begin
  S := 'Hello big wide world';
  Delete(S, 6, 9);
  WriteLn(S); //output: Hello world
end.
```

To insert a sub-string at a specified position, you can call the `Insert` procedure. You pass to this the sub-string to be inserted, the string to insert into, and the index to insert the sub-string at:

```
var
  S: string;
begin
  S := 'Goodbye world';
  Insert('cruel ', S, 9);
  WriteLn(S); //output: Goodbye cruel world
end.
```

Notice `Insert` does not overwrite any characters of the original string. If you wish to do that as well, call the `StuffString` function from `System.StrUtils`. Aside from being a function rather than a procedure, the order of this routine's parameters are slightly different to those for `Insert`: in sequence, you pass the base string, the index to insert from, the number of `Char` values to overwrite, and the new sub-string (this can be of any length):

```
var
  S: string;
begin
  S := 'It was ten years ago today';
  S := StuffString(S, 8, 3, 'twenty');
  WriteLn(S); //output: It was twenty years ago today
end.
```

Lastly, the `StringReplace` function of `System.SysUtils` replaces one or all instances of a sub-string in a main string:

```
type
  TReplaceFlags = set of (rfReplaceAll, rfIgnoreCase);

function StringReplace(const S, OldSubStr, NewSubStr: string;
  Flags: TReplaceFlags): string;
```

Be warned the `rfIgnoreCase` option isn't very intelligent. Consider the following code:

```
S := StringReplace('My favourite colour is red. Red - yes, RED!',
  'red', 'blue', [rfReplaceAll, rfIgnoreCase]);
```

What gets assigned to `S` is not `'My favourite colour is blue. Blue - yes, BLUE!'` but `'My favourite colour is blue. blue - yes, blue!'`.

The `System.StrUtils` unit also provides `ReplaceStr`, `ReplaceText`, `AnsiReplaceStr` and `AnsiReplaceText` functions. These are (very) simple wrapper functions round `StringReplace` though. As a result, `ReplaceText` and `AnsiReplaceText` (they do exactly the same thing) suffer from the same lack of intelligence shown by `StringReplace`.

Trimming whitespace from strings (Trim, TrimLeft, TrimRight)

To trim a string of whitespace at its head and tail, call `Trim`; alternatively, to trim whitespace from just the start of a string, call `TrimLeft`, and to trim it from just the end, `TrimRight`.

All these functions define 'whitespace' as characters with ordinal values of 32 or below. If you want a variant of `Trim` that takes off just ASCII space characters (i.e., #32), you'll have to roll your own:

```
function TrimSpacesOnly(const S: string): string;
var
  I, L: Integer;
begin
  L := Length(S);
  I := 1;
  while (I <= L) and (S[I] = ' ') do
    Inc(I);
  if I > L then
    Result := ''
  else
  begin
    while S[L] = ' ' do
      Dec(L);
    Result := Copy(S, I, L - I + 1);
  end;
end;
```

A variant that trims all whitespace characters defined by the Unicode standard also requires a bit of manual coding, this time using the `TCharacter` type we met earlier:

```
uses System.Character;

function UnicodeTrim(const S: string): string;
var
  I, L: Integer;
begin
  L := Length(S);
  I := 1;
  while (I <= L) and TCharacter.IsWhiteSpace(S[I]) do
    Inc(I);
  if I > L then
    Result := ''
  else
  begin
    while TCharacter.IsWhiteSpace(S[L]) do
      Dec(L);
    Result := Copy(S, I, L - I + 1);
  end;
end;
```

Splitting strings (SplitString)

Declared in `System.StrUtils`, the `SplitString` function takes a source string and a list of valid delimiters, themselves specified as a further string, and outputs a dynamic array of sub-strings. Since the array is typed to `TStringDynArray` rather than simply `TArray<string>`, you need to ensure `System.Types` is in your `uses` clause if you wish to store it.

Each delimiter is a single character. In the following code, `First`, `Second`, `Third` and `Fourth` is outputted on separate lines:

```
uses
  System.StrUtils;

var
  S: string;
begin
  for S in SplitString('First/Second/Third\Fourth', '/\') do
    WriteLn(S);
end.
```

`SplitString` employs a strict parsing algorithm, so that if there are space characters either side of a delimiter, they will be part of the returned sub-string(s). Consequently, the next snippet outputs `#One #`, `# Two #` and `# Three #`:

```
uses
  System.Types, System.StrUtils;

var
  Items: TStringDynArray;
  S: string;
begin
  Items := SplitString('One - Two - Three', '-');
  for S in Items do
    WriteLn('#' + S + '#');
end.
```

Inserting line breaks (WrapText)

The `WrapText` function of `System.SysUtils` breaks text into lines, returning a new string with added line break characters (to break up a string already containing line break characters into separate strings, use either `SplitString` or the `TStringList` class; the second is discussed below). The breaking up is done on the basis of a maximum number of `Char` values per line:

```
function WrapText(const Line, BreakStr: string;
  const BreakChars: TSysCharSet; MaxCol: Integer): string;
function WrapText(const Line: string; MaxCol: Integer): string;
```

The meaning of the parameters is as follows:

- `Line` is the string to add line breaks to.
- `BreakStr` is the characters to insert denoting a line break.
- `BreakChars` is a set of `Char` values (more exactly, `AnsiChar` values) that a line can be broken at. The idea here is to avoid breaking words into two.
- `MaxCol` is the maximum number of characters in a line before a break is required.

The simpler version of `WrapText` just delegates to the first, passing `[' ', '-', #9]` for `BreakChars` (this allows breaking on space, hyphen and tab characters), and the `SLineBreak` constant for `BreakStr`. `SLineBreak` itself (declared in the `System` unit) is defined as whatever the standard line break sequence for the platform is, which will be #13#10 (CR+LF) on Windows and #13 (CR alone) on OSX and Linux.

Case conversion (LowerCase, UpperCase, AnsiLowerCase, AnsiUpperCase)

The `LowerCase` and `UpperCase` functions perform case conversions, as their names imply. By default, only the case of characters within the ASCII range (a-z, A-Z) are altered. If you want to change all possible letters, either pass `loUserLocale` as a second parameter, or use `AnsiLowerCase`/`AnsiUpperCase` instead:

```
uses
  System.SysUtils, Vcl.Dialogs;

begin
  ShowMessage(UpperCase('café'));
  ShowMessage(UpperCase('café', loUserLocale));
  ShowMessage(AnsiUpperCase('café'));
end.
```

The first message box here will show `CAFé` (sic) while the second and third will show `CAFÉ`.

Reversing strings (ReverseString, AnsiReverseString)

The `System.StrUtils` unit contains a couple of functions for reversing a string, `ReverseString` and `AnsiReverseString`. The

difference between the two is that the first doesn't take surrogate pairs into account where the second does. In the following code, only the second call therefore performs correctly:

```
uses
  System.StrUtils, Vcl.Dialogs;

var
  S: string;
begin
  S := 'This is an example of a four byte character: 𠀀';
  ShowMessage(ReverseString(S));
  ShowMessage(AnsiReverseString(S));
end.
```

String allocation (SetLength)

In the vast majority of cases, you allocate a string simply by assigning either a string literal or any other expression that returns a string, be it another string variable, a function that returns a string, or so on. For lower level work, it can be desirable to allocate a string without specifying its contents though. When so, you can call the `SetLength` standard procedure:

```
procedure SetLength(var S: string; NewLength: Integer);
```

If the new length specified is less than the current length, the string is truncated, otherwise it is enlarged, with the added characters not being given any particular value.

If you find yourself writing a routine that creates a string from several different `Char` values, using `SetLength` and character indexing can be more efficient than repeatedly adding one `Char` after another. For example, consider these two ways to implement a function that reverses a string (the second is similar to how the `System.StrUtils` version just discussed is implemented):

```
function ReverseString1(const S: string): string;
var
  Ch: Char;
begin
  Result := '';
  for Ch in S do
    Result := Result + Ch;
end;

function ReverseString2(const S: string): string;
var
  I, Len: Integer;
begin
  Len := Length(S);
  SetLength(Result, Len);
  for I := 1 to Len do
    Result[Len - I + 1] := S[I];
end;
```

For very large strings, the second variant is ten times the speed of the first on my computer.

Beyond the freestanding routines

So far in this chapter, Delphi string handling has primarily been described in terms of standalone routines provided by the RTL. Alongside them sits a small group of classes however: `TStringBuilder`, for composing strings over multiple calls; `TStrings`/`TStringList`, for working with strings in terms of lines, keys and values; and `TRegEx`, a 'regular expressions' API for advanced pattern matching.

TStringBuilder

Declared in `System.SysUtils`, `TStringBuilder` is a utility class for building text values over a series of calls. In typical use, you create it, call `Append` and/or `AppendLine` as appropriate, then read off the completed string using the `ToString` method:

```
uses System.SysUtils;

var
  I: Integer;
  StrBuilder: TStringBuilder;
begin
  StrBuilder := TStringBuilder.Create('Here''s 1 to 4:');
  try
    StrBuilder.AppendLine;
    for I := 1 to 4 do
    begin
      StrBuilder.Append(I);
      StrBuilder.AppendLine;
    end;
    StrBuilder.Append('Z');
    StrBuilder.Append('z', 5);
    StrBuilder.Append('... I hear you say!');
    WriteLn(StrBuilder.ToString);
  finally
    StrBuilder.Free;
  end;
end.
```

This example outputs the following:

```
Here's the numbers 1 to 4:
1
2
3
4
Zzzzzz... I hear you say!
```

Constructors, properties and methods

`TStringBuilder` has the following constructors:

```
constructor Create;
constructor Create(Capacity: Integer);
constructor Create(const InitialData: string);
constructor Create(Capacity: Integer; MaxCapacity: Integer);
constructor Create(const InitialData: string; Capacity: Integer);
constructor Create(const InitialData: string;
  StartIndex, Length, Capacity: Integer);
```

If a capacity (i.e., number of `Char` values allocated up front) isn't specified, a default of only 16 is used; specify a larger value to avoid the object having to reallocate memory later (when it *does* have to reallocate, it doubles the existing capacity each time). When a maximum capacity is specified, an attempt to append more than the given number of `Char` values will raise an exception:

```
Builder := TStringBuilder.Create(10, 20);
try
  Builder.Append('This is OK'); //10 chars
  Builder.Append('So is this'); //another 10
  Builder.Append('Exception!'); //over the maximum -> error
```

The default is to not have a maximum capacity, or more exactly, a maximum capacity of `High(Integer)`.

The main properties of a `TStringBuilder` object are `Capacity`, which is read/write; `MaxCapacity`, which is read-only; `Chars`, a read/write, zero-indexed, default array property for returning individual `Char` values; and `Length`, a read/write property for truncating or extending the string being built up (if done to extend, the new characters will have undefined values):

```
var
  Builder: TStringBuilder;
begin
  Builder := TStringBuilder.Create(100, 200);
  try
    Builder.Append('Blah blah blah');
    WriteLn(Builder.Capacity);    //100
    WriteLn(Builder.MaxCapacity); //200
    WriteLn(Builder.Length);      //14
    WriteLn(Builder.Chars[0]);    //B
```

```
    Builder[7] := 'e';
    Builder[8] := 'e';
    Builder.Length := 9;
    WriteLn(Builder.ToString);    //Blah blee
  finally
    Builder.Free;
  end;
```

The public methods of `TStringBuilder` include the following:

```
function Append(const Value: <various types>): TStringBuilder;
function AppendFormat(const Format: string;
  const Args: array of const): TStringBuilder;
function AppendLine: TStringBuilder; overload;
function AppendLine(const Value: string): TStringBuilder;

procedure Clear;
procedure CopyTo(SourceIndex: Integer; const Dest: TCharArray;
  DestIndex, Count: Integer);

function Equals(StringBuilder: TStringBuilder): Boolean; reintroduce;

function Insert(Index: Integer; const <various types>): TStringBuilder;

function Remove(StartIndex, RemLength: Integer): TStringBuilder;

function Replace(const OldChar,
  NewChar: Char): TStringBuilder; overload;
function Replace(const OldValue,
  NewValue: string): TStringBuilder; overload;
function Replace(const OldChar, NewChar: Char;
  StartIndex, Count: Integer): TStringBuilder; overload;
function Replace(const OldValue, NewValue: string;
  StartIndex, Count: Integer): TStringBuilder; overload;

function ToString: string; overload; override;
function ToString(StartIndex, StrLength: Integer): string; overload;
```

`Append` and `Insert` are overloaded to take several basic types, on which the type's `xxxToStr` function or equivalent (e.g. `IntToStr`) is called and the result passed on to the main `Append` overload that takes a string. Similarly, `AppendFormat` is just a shortcut for `Append(Format(S, [arg1, arg1...]))`, and `AppendLine` just a shortcut for `Append(SLineBreak)` or `Append(S + SLineBreak)`.

All indices are zero based. Thus, if `StrBuilder.ToString` returns `'ABCDEF'`, then `StrBuilder.ToString(1, 2)` returns `'BC'`. Given Delphi string indexing is *one* based, be careful of 'off by one' bugs:

```
var
  S: string;
  Builder: TStringBuilder;
begin
  //with a string
  S := 'This is a BIG test';
  Delete(S, 11, 4);              //output: This is a test
  WriteLn(S);
  //with a string builder
  Builder := TStringBuilder.Create('This is a BIG test');
  try
    Builder.Remove(11, 4);
    WriteLn(Builder.ToString);    //output: This is a Best
  finally
    Builder.Free;
  end;
```

Use the `Equal` method if you want to check whether the content of two string builder objects is the same, since it will be slightly more performant than comparing the results of `ToString`. The test always case sensitive though, and more problematically, the respective `MaxCapacity` properties are taken into account too.

Case sensitive as well is the `Replace` method. Its four variants work as thus: (1) replaces every occurrence of one character with another; (2) replaces every occurrence of one string with another; (3) as (1), but only looking to replace existing characters within a certain range; and (4) as (2), but only looking to replace existing characters within a certain range:

```
var
  Builder: TStringBuilder;
begin
  Builder := TStringBuilder.Create;
```

```
  try
    Builder.Append('This is a big, big test');
    Builder.Replace('big', 'tiny');
    WriteLn(Builder.ToString); //output: This is a tiny, tiny test
    Builder.Clear;
    Builder.Append('This is a big, big test');
    Builder.Replace('big', 'tiny', 13, 4);
    WriteLn(Builder.ToString); //output: This is a big, tiny test
  finally
    Builder.Free;
  end;
```

Annoyingly, the versions of `Replace` that take a range will raise an exception if that range extends beyond current data:

```
  Builder := TStringBuilder.Create;
  try
    Builder.Append('She slept well: xxxxx...');
    Builder.Replace('x', 'z', 10, 100); //raises ERangeError
```

This contrasts with (for example) the `Copy` standard string function, which will silently correct overextended ranges.

TStringBuilder method style

If you look closely at the `TStringBuilder` methods, you will see that most of them are functions where you would normally have expected procedures. The reason they are functions is to enable method chaining:

```
StrBuilder.Append('One').Append('Two').Append('Three');
```

Giving Delphi isn't particularly fussy about whitespace, the above could also be written like this:

```
StrBuilder
  .Append('One')
  .Append('Two')
  .Append('Three');
```

Think twice before using this style too heavily though, since it makes debugging much harder. This is because it prevents setting a breakpoint on individual method calls — from the debugger's point of view, the three calls to `Append` here constitute a single executable step.

TStringBuilder vs. naïve string concatenation

With `TStringBuilder` available, you may be inclined to use it wherever you would have used simple string concatenation instead. In practice, doing so is unlikely to actually improve performance however. This may sound an indictment of `TStringBuilder`, but it isn't really — naïve string concatenation in Delphi is just very fast.

For example, consider the following test code. In it, naïve concatenation is performed multiple times over, without ever explicitly preallocating memory using `SetLength`:

```
const
  LoopToValue = 5500000;
  StrToConcat = 'Sometimes, doing things the easiest way ' +
                'is actually the fastest way too.';

function TestStringConcat: string;
var
  I: Integer;
begin
  Result := '';
  for I := 1 to LoopToValue do
    Result := Result + StrToConcat + sLineBreak;
end;
```

The `TStringBuilder` equivalent looks like this. We will however make its life much easier by ensuring the object has allocated every single byte of memory it will need at the off:

```
function TestStringBuilder: string;
var
  Builder: TStringBuilder;
  I: Integer;
begin
  Builder := TStringBuilder.Create(LoopToValue *
    Length(StrToConcat + sLineBreak));
  try
    for I := 1 to LoopToValue do
      Builder.AppendLine(StrToConcat);
    Result := Builder.ToString;
```

```
  finally
    Builder.Free;
  end;
end;
```

Naturally, we will need some code to perform the timings:

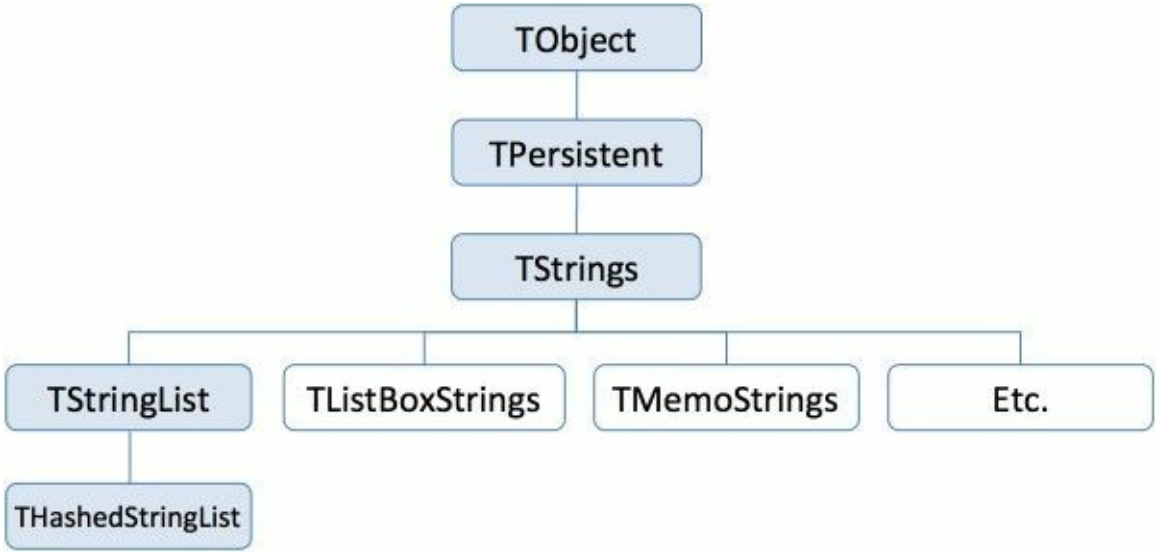
```
uses System.Diagnostics; //for TStopwatch

type
  TTestFunc = function : string;

procedure DoTiming(const Desc: string; const TestFunc: TTestFunc);
var
  Output: string;
  Stopwatch: TStopwatch;
begin
  Stopwatch := TStopwatch.StartNew;
  Output := TestFunc;
  Stopwatch.Stop;
  WriteLn(Format('%s took %n seconds creating a %d character string',
    [Desc, Stopwatch.Elapsed.TotalSeconds, Length(Output)]));
end;

begin
  DoTiming('Plus operator', TestStringConcat);
  DoTiming('TStringBuilder', TestStringBuilder);
  DoTiming('Plus operator', TestStringConcat);
  DoTiming('TStringBuilder', TestStringBuilder);
  DoTiming('Plus operator', TestStringConcat);
  DoTiming('TStringBuilder', TestStringBuilder);
end.
```

When I run this on my Windows computer, the naïve string concatenation approach beats `TStringBuilder` easily! Matters are a bit different on OS X, but even still, the gap is narrow (technically, the Delphi RTL does not employ its own memory manager on OS X, which explains the different result). Because of this, don't use `TStringBuilder` on the expectation it 'must' be faster, since it could well be slower than the more 'obvious' alternative. Rather, use it because in certain situations, it provides a more convenient interface for building up a string.

TStringList/TStrings

Defined in `System.Classes`, `TStringList` is only partly a class for creating and maintaining lists of strings. This is not to say it doesn't do that job successfully, since it does. Rather, it is because it provides lots of other functionality on top of that: in particular, it allows splitting and recombining strings in various ways, interpreting strings as key=value pairs, and loading and saving text files.

`TStringList` is structured so that the majority of its interface and even implementation appears on its parent class, `TStrings`. `TStrings` itself then appears right across the VCL and (to an extent) FireMonkey too. Thus, the `Items` property of a `TListBox`, `Lines` property of a `TMemo`, `Tabs` property of a `TTabControl`, and `Strings` property of a `TStringGrid` (amongst other examples) are all typed to it. In the case of the VCL, this produces a family tree looking like the following (FireMonkey is similar):



While `TStrings` implements the majority of the provided functionality, it leaves storage to descendant classes. With `TStringList` — usually the only descendant you explicitly instantiate — a dynamic array is used internally. In contrast, a VCL `TListBoxStrings` forwards requests to get and set items to the Windows list box API, a VCL `TMemoStrings` to the Windows multiline edit control API, and so on.

TStrings basics: adding, amending, enumerating and deleting items

To add an item to the end of a `TStrings` object, call `Add`. This method is a function that returns the new item's zero-based index (the first item having an index of 0, the second of 1, and so on). To add to a specific position instead, call `Insert`. For example, `List.Insert(0, 'new first')` will add `'hello'` to the top of the list. Also available is `AddStrings`, which copies the items from one `TStrings` to another, and `Assign` (inherited from `TPersistent`). Where `Assign` will clear the destination list before copying over items from the source, `AddStrings` won't:

```
var
  I: Integer;
  List1, List2: TStringList;
begin
  List1 := TStringList.Create;
  List2 := TStringList.Create;
  try
    //add two items to the first list
    List1.Add('First');
    List1.Add('Second');
    //add one item to the second list
    List2.Add('Zero!');
    //add the first list to the second
    List2.AddStrings(List1);
    WriteLn(List2.Count); //output: 3
    //assign the first list to the second
    List2.Assign(List1);
    WriteLn(List2.Count); //output: 2
  finally
    List1.Free;
    List2.Free;
  end;
end.
```

As shown here, the `Count` property reports the number of items currently held by the list. A writeable `Capacity` property

is also available to preallocate memory, similar to `TStringBuilder`. However, making use of it will be much less of an optimisation in the `TStringList` case because only the memory for string *references* will be preallocated, not string data.

To access or change an item previously inserted, index the object as if it were an array. Alternately, a `TStrings` object can be enumerated using a `for/in` loop:

```
var
  MyStrings: TStringList;
  S: string;
begin
  MyStrings := TStringList.Create;
  try
    MyStrings.Add('First');
    MyStrings.Add('Second');
    MyStrings[1] := 'something else'; //changes the second item
    WriteLn('The second item in the list is now "' +
      MyStrings[1] + '"');
    for S in MyStrings do
      WriteLn(S);
  finally
    MyStrings.Free;
  end;
end.
```

Once inserted, an item can be put in a different position either by calling `Move`, which takes an item's current and new index, or `Exchange`, which switches the position of two items:

```
var
  List: TStringList;
  S: string;
begin
  List := TStringList.Create;
  try
    List.Add('Wales');
    List.Add('rule');
    List.Add('the');
    List.Add('English');
    List.Move(2, 0);
    List.Exchange(1, 3);
    for S in List do
      Write(S, ' ');
  finally
    List.Free;
  end;
```

This example outputs 'the English rule Wales'.

To find an item call `IndexOf`; this returns -1 if the string cannot be found. `TStrings` objects are normally case insensitive, though `TStringList` specifically has a `CaseSensitive` property to change that:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    List.Add('First');
    List.Add('Second');
    WriteLn('Index of "second", default search: ',
      List.IndexOf('second')); //output: 1
    List.CaseSensitive := True;
    WriteLn('Index of "second", case sensitive search: ',
      List.IndexOf('second')); //output: -1
```

To remove items, call either `Delete`, which takes the position of the item to remove, or `Clear`, which removes everything. If `Delete` is called with an index that is out of bounds, an `EStringListError` will be raised:

```
List := TStringList.Create;
try
  List.Add('another');
  List.Add('test');
  List.Delete(0);
  WriteLn(List[0]); //output: test
  List.Clear;
  List.Delete(0);   //raises an exception
```

Lastly, if the `TStrings` object is tied to a control, wrap multiple updates to it in `BeginUpdate/EndUpdate` calls:

```
Memo1.Lines.BeginUpdate;
```

```
try
  Memo1.Lines.Add('Another line');
  Memo1.Lines.Add('And another one');
  Memo1.Lines.Add('And yet another');
  //...
finally
  Memo1.Lines.EndUpdate;
end;
```

This will prevent flicker, and in the case of many updates, speed things up a lot too.

Associating objects

Each item of a `TStrings` can have an associated object. For example, in a VCL application, you might have an 'owner drawn' list box in which each item has an associated `TBitmap` object.

Whatever the associated object is, it can be set either at the time the string itself is inserted through calling `AddObject` or `InsertObject` rather than `Add` or `Insert`, or later on by setting the `Objects` array property:

```
//assign associated data up front
ListBox.Items.AddObject('Second', SomeBitmap);
//assign associated data separately
ListBox.Items.Insert(0, 'First');
ListBox.Items.Objects[0] := SomeOtherBitmap;
```

If no object is explicitly associated with an item, then `nil` gets associated with it implicitly.

By default, associated objects are not considered 'owned' by the `TStrings` instance itself, meaning you must remember to explicitly free them. However, `TStringList` has an `OwnsObjects` property, which when set to `True`, will cause associated objects to be destroyed when their corresponding strings are deleted, or when the string list itself is destroyed:

```
type
  TTestObject = class
    destructor Destroy; override;
  end;

destructor TTestObject.Destroy;
begin
  WriteLn('TTestObject.Destroy');
  inherited;
end;

var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    List.OwnsObjects := True;
    List.AddObject('Test', TTestObject.Create);
  finally
    List.Free; //output: TTestObject.Destroy
  end;
```

You can change the `OwnsObjects` property at any time; what is important is the value it has at the time an item is removed or the whole list freed.

Getting and setting items from a single string (CommaText, DelimitedText, Text)

Every `TStrings` object has the ability to load or save itself to a single string via the `CommaText`, `DelimitedText` and `Text` properties. `CommaText` works in terms of sub-strings separated by commas, `DelimitedText` by sub-strings separated by some sort of character, and `Text` by sub-strings separated by line break sequences. Each of these three properties is read/write: when read, a single string is outputted, built up from the list's current contents; when written to, the list is cleared before the input string is parsed and new items added from it.

TStrings.CommaText

`CommaText` works with the format used by comma-separated values (CSV) files, a common interchange format for spreadsheet programs:

```
Smith,John,12/01/1978,Joiner
Bloggs,Joe,19/06/1976,Painter
```

Assigning the first row of such a file to `CommaText` will change the list's items to be 'Smith', 'John', '12/01/1978' and 'Joiner'.

`CommaText` treats both spaces and commas as separators, and automatically trims trailing space characters. Double quote characters act as an escaping mechanism however — wrap the whole item in them for this to work:

```
var
  List: TStringList;
  S: string;
begin
  List := TStringList.Create;
  try
    List.CommaText := 'First,"Second, I do believe",Third';
    for S in List do
      WriteLn(S);
  finally
    List.Free;
  end;
end.
```

Use of double quotes means this example outputs the following:

```
First
Second, I do believe
Third
```

TStrings.DelimitedText

For when the need to escape is overly restrictive, or alternatively, when something other than a comma should delimit items, use `DelimitedText` instead of `CommaText`. When you do, the `Delimiter`, `QuoteChar` and `StrictDelimiter` properties come into play. These default to a comma (,), double quotation mark (") and `False` respectively, matching the `CommaText` behaviour, but can be changed as desired.

When `StrictDelimiter` is set to `True`, items with spaces do not need to be wrapped in quotes, and any space at the start or end is no longer trimmed:

```
var
  List: TStringList;
  S: string;
begin
  List := TStringList.Create;
  try
    List.CommaText := ' Hello , Mr Big ';
    WriteLn(List.Count, ' items:');
    for S in List do
      WriteLn('"' + S + '" ');
    WriteLn;
    List.Delimiter := '|';
    List.StrictDelimiter := True;
    List.DelimitedText := ' Hello | Mr Big ';
    WriteLn(List.Count, ' items:');
    for S in List do
      WriteLn('"' + S + '" ');
  finally
    List.Free;
  end;
```

This outputs the following:

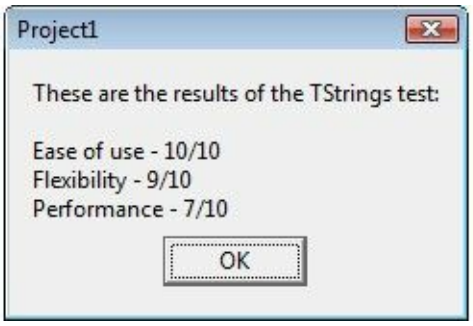
```
3 items:
"Hello"
"Mr"
"Big"

2 items:
" Hello "
" Mr Big "
```

TStrings.Text

When read, the `Text` property of a `TStrings` object returns a single string containing the list's items separated by line break sequences (by default, #13#10 on Windows and #10 on OS X). For example, the following uses a `TStringList` to build up the text for a message box:

```
uses System.Classes, Vcl.Dialogs;

var
  Strings: TStringList;
begin
```

```
  Strings := TStringList.Create;
  try
    Strings.Add('These are the results of the TStrings test:');
    Strings.Add('');
    Strings.Add('Ease of use - 10/10');
    Strings.Add('Flexibility - 9/10');
    Strings.Add('Performance - 7/10');
    ShowMessage(Strings.Text);
  finally
    Strings.Free;
  end;
end.
```



Conversely, when written to, `Text` takes the string given to it and breaks it up on the basis of line break sequences found. In the writing case, valid line break sequences are *any* of #13 alone, #10 alone, or #13#10 as a pair... or at least, that is the default situation.

The caveat is because you can change the character or characters taken to be the line break sequence by assigning the `LineBreak` property:

```
var
  List: TStringList;
  S: string;
begin
  List := TStringList.Create;
  try
    List.LineBreak := 'BOOM';
    List.Text := 'FirstBOOMsecondBOOMthird' + SLineBreak +
      'still third';
    WriteLn('READ IN');
    WriteLn('Count = ', List.Count, ':');
    for S in List do
      WriteLn('"' + S + '"');
    WriteLn;
    WriteLn('OUTPUT');
    List.LineBreak := ' - ';
    WriteLn('"' + List.Text + '"');
  finally
    List.Free;
  end;
end.
```

If you run this example, the following is outputted:

```
READ IN
Count = 3:
"First"
"second"
"third
still third"

OUTPUT
"First - second - third
still third - "
```

After customising `LineBreak`, sanity may be restored by assigning the global `SLineBreak` constant to it:

```
List.LineBreak := SLineBreak;
```

Loading and saving text files or streams (LoadFromXXX, SaveToXXX)

To load the contexts of a text file into a `TStrings` object, call `LoadFromFile`; with the contents of a stream, `LoadFromStream`. To write the contents of a `TStrings` instance to a text file, call `SaveToFile`, and to a stream, `SaveToStream`:

```
Memo.Lines.LoadFromFile('C:\Users\CCR\My source file.txt');
```

```
Memo.Lines.SaveToFile('C:\Users\CCR\My dest file.txt');
```

When `SaveToFile` is passed the name of a file that already exists, the previous contents of the file will be overwritten rather than appended to.

So far so straightforward, but there is an important complication: text can be 'encoded' in multiple ways, and the native encoding of Delphi strings (UTF-16) is be an unpopular one for text files, notwithstanding the fact it is used as an in-memory format by many other programming environments too. In fact, it is so unpopular as a saving format that neither `LoadFromXXX` nor `SaveToXXX` use it by default! On Windows, the legacy 'Ansi' encoding will serve that role instead; on OS X, UTF-8. The Windows case may sound bizarre — why write files in a legacy encoding by default? — but it is the Windows norm, as exhibited by (for example) Notepad.

Nonetheless, this may cause issues if the `TStrings` contains 'exotic' characters. When so, you should use a suitable Unicode encoding, such as UTF-8. This can be done by passing `TEncoding.UTF8` to `SaveToFile` or `SaveToStream` as a second parameter:

```
Strings.SaveToFile('C:\Users\CCR\My file.txt', TEncoding.UTF8);
```

Alternatively, the `DefaultEncoding` property can be set beforehand:

```
Strings.DefaultEncoding := TEncoding.UTF8;
Strings.SaveToStream(SomeStream);
```

Saving a file like this will include a 'byte order mark' (BOM) by default. In the case of UTF-8 on Windows, this is generally desirable. Moreover, it means `LoadFromFile` will be able to read the file back in properly without any special handling on your part. Nonetheless, if you don't want a BOM outputted, just set the `WriteBOM` property to `False` before calling `SaveToXXX`. Thus, a routine for appending UTF-8 encoded text to an existing file might be written like this:

```
procedure AppendUTF8Text(Source: TStrings; const Dest: string);
var
  SavedWriteBOM: Boolean;
  Stream: TFileStream;
begin
  SavedWriteBOM := Source.WriteBOM;
  Stream := TFileStream.Create(Dest, fmOpenWrite);
  try
    Stream.Seek(0, soEnd);
    Source.WriteBOM := False;
    Source.SaveToStream(Stream, TEncoding.UTF8);
  finally
    Source.WriteBOM := SavedWriteBOM;
    Stream.Free;
  end;
end;
```

Correspondingly, both `LoadFromFile` and `LoadFromStream` are overloaded to take an explicit `TEncoding` parameter just like `SaveToFile` and `SaveToStream`. When set, `TStrings` will *always* assume the specified encoding is in effect, skipping its BOM (if applicable) when found, but not raising an error if it isn't:

```
Strings.LoadFromFile('My file.txt', TEncoding.UTF8);
```

Once more, the alternative to passing an explicit `TEncoding` parameter is to set the `DefaultEncoding` property beforehand:

```
Strings.DefaultEncoding := TEncoding.UTF8;
Strings.LoadFromFile('C:\Users\CCR\My file.txt');
```

In this case, UTF-8 will be assumed *if* no valid BOM is found. While individual circumstances are inevitably different, the chances are this will be the preferred behaviour.

Name=value indexing

All `TStrings` instances support simple name=value indexing. For example, imagine adding the following items to a string list: `'first=most important'`, `'second=not so important'` and `'third=utterly insignificant'`. Once done, the `Values` array property then allow accessing the part after the equals sign by reference to the part prior to the equals sign. If the 'name' specified isn't found, then an empty string is returned for its 'value':

```
var
  Strings: TStringList;
begin
  Strings := TStringList.Create;
  try
    Strings.Add('first=most important');
    Strings.Add('second=not so crucial');
    Strings.Add('third=utterly insignificant');
```

```
    Writeln(Strings.Values['second']);  //not so crucial
    Writeln(Strings.Values['first']);   //most important
    Writeln(Strings.Values['rubbish']); //(an empty line)
  finally
    Strings.Free;
  end;
end.
```

The `Values` property is also writeable; when the name specified does not exist, a new line gets added to the list:

```
List := TStringList.Create;
try
  List.Values['New stuff'] := 'Sparkling';
  WriteLn(List.Text); //output: New stuff=Sparkling
```

To access a value by index and not name, use `ValueByIndex`; similarly, a `Names` array property allows accessing the name part of a name/value pair by its index. Lastly, a `NameValueSeparator` property is available if you wish to replace the equals sign with some other name/value delimiter:

```
List.NameValueSeparator := '|';
List.Add('First born|Exiting');
List.Add('Second born|Contended');
List.Add('Third born|Regretful');
WriteLn(List.Names[1], ' = ',
  List.ValueFromIndex[1]); //output: Second born = contented
```

In practice, using the `TStrings` name/value feature on large datasets won't be very performant. Nonetheless, if you expect to load a string list up front before mostly just reading values from it, then the `THashedStringList` class, declared in `System.IniFiles`, will provide an optimised implementation of the standard `TStrings` interface. Otherwise, the generic `TDictionary` class (covered in the next chapter) is likely to be a better choice, albeit at the cost of being a specialist name/value class rather than a `TStrings` descendant.

TStringList specifics

Almost all the things looked at in this section so far apply to any `TStrings` object. A few features are unique to `TStringList` specifically however. The main one is sorting, and related to that, faster searching. The interface looks like the following:

```
type
  TDuplicates = (dupIgnore, dupAccept, dupError);
  TStringListSortCompare = function(List: TStringList;
    Index1, Index2: Integer): Integer;

function Find(const S: string; var Index: Integer): Boolean;
procedure Sort; virtual;
procedure CustomSort(Compare: TStringListSortCompare); virtual;
property Duplicates: TDuplicates read FDuplicates write FDuplicates;
property Sorted: Boolean read FSorted write SetSorted;
```

Either `Sort` or `CustomSort` can be called at any time. In the case of `CustomSort`, a standalone function must be passed that returns a positive number if the item at `Index1` should come after the item at `Index2`, a negative number if the item at `Index2` should come after the one at `Index1`, or 0 if the two items should be considered equal. Use of the `CompareText` or `CompareStr` functions can be useful in implementing the callback function:

```
function SortDescendingCallback(List: TStringList;
  Index1, Index2: Integer): Integer;
begin
  Result := -CompareText(List[Index1], List[Index2], loUserLocale);
end;

var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    List.Add('First');
    List.Add('Second');
    List.Add('Third');
    //sort the list in reverse alphabetical order
    List.CustomSort(SortDescendingCallback);
```

When the `Sorted` property is set to `True`, `Sort` is called and the list kept sorted thereafter. Furthermore, the `Duplicates` property comes into play: when set to `dupIgnore` (the default), attempts to add an item that already exists will be ignored; when set to `dupAccept`, it will be added as normal; and when set to `dupError`, an `EStringListError` exception will be raised:

```
var
```

```
  List: TStringList;
begin
  List := TStringList.Create;
  try
    List.Sorted := True;
    List.Add('boys');
    List.Add('girls');
    List.Add('boys');
    WriteLn(List.Count);          //output: 2
    List.Duplicates := dupAccept;
    List.Add('boys');
    WriteLn(List.Count);          //output: 3
    List.Duplicates := dupError;  //does NOT raise an exception
    List.Add('boys');             //does now however
  finally
    List.Free;
  end;
```

As shown here, setting `Duplicates` to `dupError` when duplicates already exist does not cause an exception — the setting only takes effect when you attempt to add further items.

A further effect of `Sorted` being `True` is that the `Find` method now becomes operable:

```
var
  Index: Integer;
  List: TStringList;
begin
  //...
  if List.Find('something', Index) then
    WriteLn('"something" is at position ', Index)
  else
    Writeln('"something" could not be found');
```

Since `Find` makes use of the fact the list is sorted, it can be much speedier than `IndexOf` when the latter is called against an unsorted list (when `Sorted` is `True`, `IndexOf` simply delegates to `Find`).

Note that all of `Sort`, `Sorted`, `Duplicates` and `Find` respect the `CaseSensitive` property. However, changing `CaseSensitive` while `Sorted` is `True` doesn't cause a resort — if you want that, you must toggle `Sorted` to `False` and back to `True` again afterwards.

Observing updates

Another object can observe changes to a `TStringList` by handling one or both of its `OnChanging` and `OnChange` events. Both are typed to `TNotifyEvent`, and so take a method with a single `TObject` parameter representing the sender — the string list in this case. `OnChanging` is called immediately prior to a change and `OnChange` immediately afterwards. For example, if the class `TTest` has a private `TStringList` field called `FStrings`, it might define a procedure called `StringsChanged` and assign it to the string list's `OnChange` event as so:

```
type
  TTest = class
  strict private
    FStrings: TStringList;
    procedure StringsChanged(Sender: TObject);
  public
    constructor Create;
    destructor Destroy; override;
    property Strings: TStringList read FStrings;
  end;

constructor TTest.Create;
begin
  inherited Create;
  FStrings := TStringList.Create;
  FStrings.OnChange := StringsChanged;
end;
```

Handling `OnChange` may be useful here given the string list is exposed as a property, and so, may have its contents changed by external code.

Regular expressions

'Regular expressions' ('regex' for short) are a way to search for patterns inside a piece of text. Using them, you can validate whole strings, extract sub-strings, and perform find-and-replace operations, all on the basis of more or less complex patterns rather than the simple character indices you would have to use otherwise.

Some form of regex support is found right across most modern programming environments. This means the regular expressions you might use in a Delphi application can be applied elsewhere too, and vice versa. Put another way, the regex syntax forms its own mini-programming language that stands apart from the particular programming environment it is used in.

Nonetheless, this syntax is not standardised, resulting in many different dialects existing — the JavaScript one is not quite the same as the Java one, which is not quite the same as the .NET one and so on. Delphi supports the PCRE ('Perl Compatible Regular Expressions) dialect — in fact, PCRE itself is used under the bonnet. This can be useful to know if you come across an example of a complex regular expression that you would like to use for your own applications. (As an aside, PCRE is to the actual Perl dialect what American English is to British English, roughly speaking, though with a completely independent implementation.)

In Delphi, the interface for working with regular expressions comes in three layers, spread over three units. The lowest level (`System.RegularExpressionsAPI`) is the interface for the raw PCRE API, and not something you would usually use directly. The next level (`System.RegularExpressionsCore`) constitutes a class- and `UTF8String`-based wrapper of the PCRE interface (`UTF8String` not `string`, since PCRE uses UTF-8 rather than UTF-16 internally). This level can be appropriate if you are able to work with `UTFString` more generally. On top of it is layered the `System.RegularExpressions` unit. Aside from using `string` not `UTF8String`, this surfaces its functionality as a series of record-based object types, easing memory management. Unless you have good reason not to, this is the most appropriate level to work at, and is what will be described here.

TRegEx

The central type of `System.RegularExpressionsCore` is `TRegEx`. This is a record you can either explicitly instantiate or call static methods on, in which case a temporary instance will be constructed internally. To explicitly instantiate, call the `Create` constructor, passing the pattern to match and (optionally) a set of options; being a record, you do not call `Free` when you have finished with it. When using the static methods, the pattern and (if applicable) option set are passed in as extra parameters:

```
uses
  System.SysUtils, System.RegularExpressions;

var
  RegEx: TRegEx;
  Success: Boolean;
begin
  { Call static method, implicitly creating an instance }
  Success := TRegEx.IsMatch('I am HAPPY', 'happy',
    [roIgnoreCase]);                          //True
  { Explicitly create an instance }
  RegEx := TRegEx.Create('happy', [roIgnoreCase]);
  Success := RegEx.IsMatch('I am HAPPY');    //True
end.
```

If you are using the same expression on lots of input strings, explicitly instantiating `TRegEx` and using the instance methods will give your code a performance boost, however it doesn't really matter otherwise.

Matching methods

The simplest method available is `IsMatch`. This tests a string against a given expression, returning `True` if it is found and `False` otherwise. If you require details about the actual match, call either `Match` or `Matches` instead. `Match` returns a `TMatch` record with details of the first match — call its `NextMatch` method to return details of the next one. In contrast, `Matches` finds all matches up front, before returning a `TMatchCollection` record you can enumerate:

```
uses System.RegularExpressions;

var
  I: Integer;
  Item: TMatch;
  Matches: TMatchCollection;
begin
  { Enumerate one by one }
```

```
  Item := TRegEx.Match('Liam Bonehead Noel Guigsy Tony',
    '(Liam|Noel)');
  while Item.Success do
  begin
    WriteLn(Item.Value);
    Item := Item.NextMatch;
  end;
  { Enumerate using Matches }
  for Item in TRegEx.Matches('tea coffee cocoa', 'co[a-z]+') do
    WriteLn(Item.Value);
  { Get a TMatchCollection explicitly }
  Matches := TRegEx.Matches('the colours were very colorful',
   'colou?r');
  WriteLn(Matches.Count, ' matches:');
  for I := 0 to Matches.Count - 1 do
    Write(' ' + Matches[I].Value);
```

This outputs the following:

```
Liam
Noel
coffee
cocoa
2 matches: colour color
```

The public interface of `TMatchesCollection` is very simple, being composed of an `Item` default property that returns a `TMatch` instance by its zero-based index, a `Count` property to report how many matches there are, and a `GetEnumerator` method to support `for`/`in` loops. In the preceding code, the first `Matches` call makes use of the enumerator, and the second the default property.

The public interface of `TMatch` is a bit more elaborate, but not much:

```
function NextMatch: TMatch;
function Result(const Pattern: string): string;
property Groups: TGroupCollection read GetGroups;
property Index: Integer read GetIndex;
property Length: Integer read GetLength;
property Success: Boolean read GetSuccess;
property Value: string read GetValue;
```

For a `TMatch` object returned by a `TMatchCollection`, `Success` will always return `True`, otherwise it indicates whether the previous match (if any) was the last one. `Index` returns the one-based index of the match in the source string, and `Length` the number of characters in the match. For example, the `TMatch` object returned by `TRegEx.Match('my favorite', 'favou?rite')` will report 4 for `Index` and 8 for `Length`. Of the other members, `Result` will transform the match according to the specified replacement pattern, and `Groups` breaks the match into sub-matches. We will look at both shortly.

Elements of a regular expression

Matching expressions are potentially composed of 'literals', 'character sets', 'alternators', 'character classes', 'quantifiers', 'anchors' and 'lookarounds', along with 'groups' and 'backreferences'.

A **literal** is simply a character that *must* be found for the match as a whole to succeed. A matching expression such as `'red'` is composed solely of literals, and when used, turns the regular expression engine into a glorified `Pos` function:

```
I := Pos('red', 'the colour is red');               //15
I := TRegEx.Match('the colour is red', 'red').Index; //15
```

Because they have special meaning, certain characters (called 'metacharacters' in the regex jargon) must be 'escaped' with a preceding backslash to be used as a literal: `*`, `+`, `?`, `|`, `{`, `[`, `(`, `)`, `^`, `$`, `.`, `#` and the backslash itself, `\`. Consequently,

```
TRegEx.IsMatch('Is that a question?', '?')
```

will raise an exception, but

```
TRegEx.IsMatch('Is that a question?', '\?')
```

will return `True`.

Character sets enable matching to one of a group of characters. A set is defined by placing the characters concerned inside square brackets. Thus, `[ab]` will match against both `'a'` and `'b'`. A hyphen can be used to specify a range, for example `[1-9]` will match against any a positive digit, and multiple ranges can be included in the same set, e.g. `[a-cM-O]` will match any one of `'a'`, `'b'`, `'c'`, `'M'`, `'N'` and `'O'`.

The `^` symbol is also available to require any character *except* one in the set: `'[^euioa]'` therefore matches any character that is not a lower-case vowel.

Alternators have a similar effect to character sets, only with respect to allowing for alternate sub-strings rather than alternate characters. The syntax is to put the acceptable variants in round brackets, delimited by a vertical bar. Thus, `(red|blue)` matches against both `'red'` and `'blue'`.

A **quantifier** requires the preceding item to appear a certain number of times:

- `?` matches zero or one time
- `*` matches any number of times
- `+` matches 1 or more times
- `{n}` (where `n` is an integer) requires the item to repeat the specified number of times
- `{n,}` requires the item to repeat at least `n` times
- `{n,m}` requires the item to repeat between `n` and `m` times

```
TRegEx.IsMatch('favorite', 'favou?rite');             //True
TRegEx.IsMatch('favorite', 'favou+rite');             //False
TRegEx.IsMatch('He snoozed: zzz...', ' z{4,5}\.');    //False
TRegEx.IsMatch('He snoozed: zzzz...', ' z{4,5}\.');   //True
TRegEx.IsMatch('He snoozed: zzzzzz...', ' z{4,5}\.'); //False
```

If the 'item' should be a substring rather than just a single character, enclose it in brackets:

```
TRegEx.IsMatch('They are playing', 'play(ing)?');       //True
TRegEx.IsMatch('They have played', 'play(ing)?');       //True
TRegEx.IsMatch('She was wearing plaits', 'play(ing)?'); //False
```

Quantifiers are by default 'greedy' in that they will chomp through as much as they can:

```
S := TRegEx.Match('The cow went moooo!', 'o{2,3}').Value; //ooo
S := TRegEx.Match('The cow went moooo!', 'o{2,}').Value;  //oooo
```

The converse of a greedy quantifier is a 'lazy' one, in which the smallest possible match is taken. To make a specific quantifier lazy, suffix it with a question mark:

```
S := TRegEx.Match('The cow went mooooo!', 'o{2,}?').Value; //oo
```

A **character class** is in effect a predefined character set:

- `.` (i.e., a full stop/period) matches *any* character but for `#13` and `#10`:

```
S := TRegEx.Match('1st'#13#10'2nd', '.*').Value; //1st
```

An easy mistake to make with `.` is to forget quantifiers are greedy by default:

```
S := TRegEx.Match('stuff! more stuff!', '.*\!').Value;
  //stuff! more stuff!
S := TRegEx.Match('stuff! more stuff!', '.*?\!').Value;
  //stuff!
```

- `\d` matches any digit — this has the same effect as `[0-9]`
- `\s` matches any ASCII whitespace character
- `\n` matches the ASCII 'line feed' character, which is equivalent to embedding #10 in the matching expression, i.e. `IsMatch(S, '.*'#10)` is equivalent to `IsMatch(S, '.*\n')`.
- `\r` matches the ASCII 'carriage return' character, which is equivalent to embedding #13 in the matching expression.
- `\t` matches the ASCII 'tab' character, which is equivalent to embedding #9 in the matching expression.
- `\w` matches a 'word' character, defined as `[0-9a-zA-Z]`

```
TRegEx.IsMatch('A1!', '\w\d.'); //True
TRegEx.IsMatch('A1', '\w\d.');  //False (no 3rd char)
TRegEx.IsMatch('AA!', '\w\d.'); //False (2nd char not a digit)
TRegEx.IsMatch('-1!', '\w\d.'); //False (1st char neither a
                                //letter nor a digit)
```

- `\p{c}` matches a character of the specified Unicode category, where c is `L` for letters, `N` for numbers, `P` for punctuation, and so on (a fuller list will be given shortly). Where `\w` will not pick up accented letters, Cyrillic letters, and so on, `\p{L}` will.

For any character class specified with a slash followed by a letter code, putting the letter in upper case causes the match to be negated, e.g. `\D` means anything *but* a digit:

```
Success := TRegEx.IsMatch('8', '\D'); //False
```

The category code casing for `\p` is fixed however, so that `\p{N}` picks up a number and `\P{N}` picks up anything but a number.

Anchors restrict matches to certain positions:

- `^` matches to the start of the input string, assuming multiline mode is not set.
- `$` matches to the end of the input string, assuming multiline mode is not set.
- `\A` matches to the start of the input string regardless of multiline being set.
- `\Z` matches to the end of the input string regardless of multiline being set.
- `\b` matches to the start or end of a 'word'. Like `\w`, this is ASCII only, so that `TRegEx.Match('café', '\b.+\b').Value` will return just `'caf'` given `'é'` is outside the ASCII range.
- `\G` restricts multiple matches to that a subsequent match is only valid if it begins immediately after its predecessor in the input string—

```
  Num := TRegEx.Matches('Baden-Baden', 'Baden').Count;   //2
  Num := TRegEx.Matches('Baden-Baden', '\GBaden').Count; //1
```

When the 'multiline' option is set, the behaviour of `^` and `$` changes to require matching to the start and end of a line respectively:

```
const
  TestStr = 'First' + SLineBreak + 'Second';
var
  Match: TMatch;
begin
  Write('Without setting roMultiLine:');
  for Match in TRegEx.Matches(TestStr, '^.*') do
    Write(' ' + Match.Value); //output: First
  WriteLn;
  Write('With roMultiLine set:');
  for Match in TRegEx.Matches(TestStr, '^.*', [roMultiLine]) do
    Write(' ' + Match.Value); //output: First Second
```

`\A` and `\Z` are unaffected however.

Lookaround assertions (also called 'zero-width assertions') are a generalisation of anchors in which the 'anchor' is custom defined. In each case, what is returned as the match is not altered — the lookaround assertion just restricts the *context* in which a match is valid:

- `(?=`*expr*`)` where *expr* is a valid matching expression defines a 'positive lookahead', in which the secondary expression must be found **after** the main one —

```
  var
    Match: TMatch;
  begin
    //without a lookahead
    Match := TRegEx.Match('test test!', 'test');
    WriteLn(Match.Value);                    //output: test
    WriteLn(Match.Index);                    //output: 1
    //require a trailing exclamation mark
    Match := TRegEx.Match('test test!', 'test(?=\!)');
    WriteLn(Match.Value);                   //output: test
    WriteLn(Match.Index);                   //output: 6
```

- `(?!`*expr*`)` where *expr* is a valid matching expression defines a 'negative lookahead', in which the secondary expression must **not** be found **after** the main one —

```
  var
    Match: TMatch;
  begin
    { Without a lookahead }
    Match := TRegEx.Match('loser loses', 'lose');
    WriteLn(Match.Value);                       //output: lose
    WriteLn(Match.Index);                       //output: 1
    { With a lookahead (no 'r' may immediately follow) }
    Match := TRegEx.Match('loser loses', 'lose(?!r)');
    WriteLn(Match.Value);                       //output: lose
    WriteLn(Match.Index);                       //output: 7
```

- `(?<=`*expr*`)` where *expr* is a valid, fixed length matching expression defines a 'positive lookbehind', in which the

secondary expression must be found **before** the main one.

- `(?<!expr)` where *expr* is a valid, fixed length matching expression defines a 'negative lookbehind', in which the secondary expression must **not** be found **before** the main one —

```
TRegEx.IsMatch('Joe has no hope', 'hope');         //True
TRegEx.IsMatch('Joe has no hope', '(?<!no )hope'); //False
```

The fact lookbehind (unlike lookahead) matching expressions must be fixed length means you are restricted to literals, character sets, character classes and the `{n}` quantifier within the sub-expression.

Unicode: using \p{c}

While the PCRE engine used by `TRegEx` does understand Unicode, it remains ASCII only in its treatment of `\d`, `\s`, `\w` and `\b`. To pick out non-ASCII characters, you must therefore use the `\p{c}` syntax, where c is one of the following:

- `L` for letters; use `Lu` and `Ll` for upper and lower case letters specifically. Thus, `\p{Lu}` will match the capital eszett, ẞ, and `\p{Lu}` will match the normal eszett, ß.
- `N` for numbers; use `Nd` for decimal numbers specifically, and `Nl` for letter numbers, e.g. Roman numerals.
- `S` for symbols; use `Sc` for currency symbols specifically (e.g. £), and `Sm` for mathematical symbols (e.g. ÷).
- `P` for punctuation
- `Z` for separators
- `M` for diacritic marks
- `C` for control characters

The meaning of 'number', 'symbol' etc. is as per the Unicode standard, and corresponds to what `TCharacter.GetUnicodeCategory` will return.

Using the `\p` syntax, a Unicode-aware substitute for `\w` would be `(\p{L}|\p{N})`, i.e. an alternator that selects either a letter or a number:

```
var
  Match: TMatch;
begin
  for Match in TRegEx.Matches('Café Motörhead', '\w+') do
    WriteLn(Match.Value);
  for Match in TRegEx.Matches('Café Motörhead', '(\p{L}|\p{N})+') do
    WriteLn(Match.Value);
```

Where the first attempt in this example outputs `Caf`, `Mot` and `rhed`, the second correctly outputs `Café` and `Motörhead`.

Similar to the other character class codes, using an upper case 'P' — i.e. `\P{c}` — matches to anything but the specified category. For example, `\P{N}` matches against `'V'` (the letter) but not `'Ⅴ'` (the Roman numeral — `#$2164`).

Capturing groups

By default, every time an expression appears in round brackets and a question mark does not immediately follow the opening bracket, an implied 'capturing group' is created. When the expression as a whole succeeds, each group can be accessed via the `Groups` property of the `TMatch` record:

```
var
  Group: TGroup;
  Match: TMatch;
begin
  Match := TRegEx.Match('hello world', '(.)orl(.)');
  WriteLn('Whole match: "', Match.Value, '"');
  WriteLn('No. of groups: ', Match.Groups.Count);
  for Group in Match.Groups do
    WriteLn(Group.Value);
```

This produces the following output:

```
Whole match: "world"
No. of groups: 3
world
w
d
```

As this example illustrates, the first item returned by the `Groups` property represents the whole match, with subsequent

items representing the sub-matches corresponding to the bracketed sub-expressions.

The `Groups` property itself is typed to `TGroupCollection`, a simple record that implements an enumerator for `for/in` support, a `Count` property for reading the number of items, and an `Items` default array property for reading off an item by its zero-based index (given the first item represents the whole match, the first captured group has an index of 1). A group itself is represented by a `TGroup` record, which is a simplified version of `TMatch` that contains `Index`, `Length`, `Success` and `Value` properties:

```
var
  Group: TGroup;
begin
  Group := TRegEx.Match('hello world', '(.)orl(.)').Groups[2];
  WriteLn(Group.Index);   //output: 11
  WriteLn(Group.Length);  //output: 1
  WriteLn(Group.Success); //output: TRUE
  WriteLn(Group.Value);   //output: d
```

Similar to `TMatch`, `Index` here means the character index of the sub-match in the input string, not the index of the group itself.

Backreferences

Aside from being accessible from the `TMatch` record, captured group sub-matches can also be accessed within the matching expression itself. This is done using the 'backreference' syntax, which takes the form of a backslash followed by a digit, where `\1` references the first captured group, `\2` the second and so on:

```
S := TRegEx.Match('Free dosh - quick Jonno, quick!',
  '(\w{5}).*\1').Value;
```

Here, the captured group requests a continuous sequence of five ASCII 'word' characters, matching to `'quick'` in the input string. The next part of the expression then matches anything up to and including the next time the same sequence is found, resulting in `'quick Jonno, quick'` being assigned to `S`.

Note that if you do not require backreference functionality for a specific bracketed sub-expression, you can let the regex engine know by placing a question mark and a colon immediately after the opening bracket, e.g. `(?:foo|bar)` instead of just `(foo|bar)`. This will tell the engine not to bother saving the sub-match. If you do not require backreference functionality for *any* group, then you can also disable it globally using the `roExplicitCapture` option:

```
GotIt := TRegEx.IsMatch('it is red', 'is (red|blue)',
  [roExplicitCapture]);
```

Naming captured groups

As such, captured groups are normally anonymous, being identified by an implicit index. You can however name groups as well. When naming a group, use the syntax `(?P<name>expr)`; to reference a named group inside the expression, use `(?P=name)`, and in Delphi code, use the `Groups` property as before, only passing the name as the index:

```
var
  Match: TMatch;
  S1, S2; string;
begin
  Match := TRegEx.Match('red foo bar blee red oh',
    '(?P<color>red|blue).*(?P=color)');
  S1 := Match.Value;                 //red foo bar blee red
  S2 := Match.Groups['color'].Value; //red
```

Named groups are still implicitly indexed by number, making numeric referencing possible both within the expression and from the `Groups` property. However, they are unaffected by the `roExplicitCapture` option. The following code therefore does the same thing as the previous example:

```
var
  Match: TMatch;
  S1, S2; string;
begin
  Match := TRegEx.Match('red foo bar blee red oh',
    '(?P<color>red|blue).*\1', [roExplicitCapture]);
  S1 := Match.Value;                 //red foo bar blee red
  S2 := Match.Groups[1].Value;       //red
```

Specifying options

In all, the `TRegExOptions` set is composed of the following possible elements to tweak the default behaviour of the regex

engine:

- `roCompiled` causes the PCRE engine to transform the expression into its internally parsed form immediately rather than when `Match`, `Matches` or `IsMatch` is first called. This option is irrelevant if you don't explicitly instantiate `TRegEx` instance.
- `roExplicitCapture` causes capturing groups to be created only when they are explicitly named.
- `roIgnoreCase` causes case insensitive matching (the default is case sensitive). This option is Unicode aware:

```
TRegEx.IsMatch('ПРИВЕТ', 'привет');                 //False
TRegEx.IsMatch('ПРИВЕТ', 'привет', [roIgnoreCase]); //True
```

- `roIgnorePatternSpace` causes whitespace (including line break sequences) inside a matching expression to be ignored rather than treated as literal characters. It also allows descriptive comments to be embedded in the matching expression — prefix text with a hash symbol (`#`), and the rest of the line will be ignored by the regex engine:

```
{ Comment without roIgnorePatternSpace causes match to fail }
GotIt := TRegEx.IsMatch('red blue green',
  '#this is a comment' + sLineBreak + 'blue');
{ Comment with roIgnorePatternSpace allows match to succeed }
GotIt := TRegEx.IsMatch('red blue green',
  '#this is a comment' + sLineBreak + 'blue', [roIgnorePatternSpace]);
```

- `roMultiLine` changes the meaning of ^ and $ to match against the start and end of lines rather than the whole input string.
- `roSingleLine` changes the meaning of . to match with any character, rather than any character but for `#13` and `#10`.

Splitting strings

Using the same expression syntax used by `IsMatch`, `Match` and `Matches`, you can split a string into a dynamic array of sub-strings via the `Split` method:

```
class function Split(const Input, Pattern: string): TArray<string>;
class function Split(const Input, Pattern: string;
  Options: TRegExOptions): TArray<string>; static;
function Split(const Input: string): TArray<string>;
function Split(const Input: string; Count: Integer): TArray<string>;
function Split(const Input: string; Count,
  StartPos: Integer): TArray<string>;
```

`Input` is split on the basis of `Pattern` (or in the case of the instance methods, the expression string that was passed to `Create`), which determines what should be treated as a delimiter. In contrast to both the `Split` standalone function and `TStrings.DelimitedText` property we met earlier, the delimiter doesn't have to be a single character:

```
var
  S: string;
begin
  for S in TRegEx.Split('theaaatimeaaaisaaanow', 'aaa') do
    Write(S + ' '); //output: the time is now
```

In principle, `Match` or `Matches` can be used to split a string too. When doing that though, the matching pattern would specify what is *not* a delimiter. Invert the pattern, and you have an expression suitable for a `Split` call. The following example demonstrates this, since the `Matches` and `Split` calls extract the same things (`'red'`, `'blue'` and `'green'`):

```
var
  Match: TMatch;
  S: string;
begin
  for Match in TRegEx.Matches('red blue green', '\w+') do
    WriteLn(Match.Value);
  for S in TRegEx.Split('red blue green', '\W+') do
    WriteLn(S);
```

Use the more complicated instance method versions of `Split` if you want to put a cap on the number of splittings, or start splitting from somewhere other than the start of the input string:

```
var
  RegEx: TRegEx;
  S: string;
begin
  RegEx := TRegEx.Create('\W+');
  for S in RegEx.Split('red blue green purple cyan', 3, 5) do
    WriteLn(S);
```

Given the cap on 3 matches, and the fact the input string is split only from the fifth character, this outputs the following:

```
red blue
green
purple cyan
```

Replacing sub-strings

Aside from matching and splitting, `TRegEx` also supports replacing sub-strings. The `Replace` method has four variants that come in both instance and static method forms. The instance method versions are like this (the static methods follow the same pattern):

```
type
  TMatchEvaluator = function(const Match: TMatch): string of object;

function Replace(const Input, Replacement: string): string;
function Replace(const Input: string;
  Evaluator: TMatchEvaluator): string;
function Replace(const Input, Replacement: string;
  Count: Integer): string;
function Replace(const Input: string;
  Evaluator: TMatchEvaluator; Count: Integer): string;
```

The content of `Replacement` has its own syntax. In the simplest case, it is composed solely of literals that require no escaping:

```
S := TRegEx.Replace('Delphi XE2, C++Builder XE2', '2', '3');
```

This sets `S` to `'Delphi XE3, C++Builder XE3'`. Either a backslash or a dollar sign introduce special values:

- `\&`, `\0`, `$&` and `$0` all output what was matched

```
  S := TRegEx.Replace('123 45 6', '\d+', '\&0'); //1230 450 60
```

- `\1` through `\99`, `$1` through `$9`9, and `${1}` through `${99}`, are all valid ways to reference a captured group by its index

```
  S := TRegEx.Replace('99 not 1',
   '(\d+) not (\d+)', '$2 not $1'); //1 not 99
```

- `\g<name>` and `${name}` both reference a named captured group

```
  S := TRegEx.Replace('911 not 999',
    '(?P<first>\d+) not (?P<second>\d+)',
    '\g<second> not ${first}');          //999 not 911
```

- `\`` or `$`` (backtick) references the input string to the left of the match

```
  S := TRegEx.Replace('no yes', '\w{3}', '\`'); //no no
```

- `\'` or `$'` (straight single quote) references the input string to the right of the match

```
  S := TRegEx.Replace('no yes', '\b\w\w\b', '$'''); // yes yes
```

- `$_` outputs the entire input string

```
  S := TRegEx.Replace('repetition is boring - OK', 'OK', '$_');
  //repetition is boring - repetition is boring - OK
```

If there is not any ambiguity, both single dollar signs and backslashes can be used as literals, however it is safer to double them up:

```
  S := TRegEx.Replace('£8.80', '£', '$$'); //$8.80
```

Replacing with a callback function

Instead of providing a replacement expression, you can provide a replacement function. This function must be a normal method (it can't be an anonymous method) that takes a `TMatch` record and returns a string, though a class method can be used if you would prefer not to instantiate anything. The following example provides a callback function in order to implement a `TitleCase` function via `TRegEx`:

```
uses
  System.SysUtils, System.Character, System.RegularExpressions;

type
  TMyUtils = class
  strict private
    class function Callback(const AMatch: TMatch): string;
  public
    class function TitleCase(const S: string): string;
  end;
```

```
class function TMyUtils.Callback(const AMatch: TMatch): string;
begin
  Result := AMatch.Value;
  Result := ToUpper(Result[1]) + Copy(Result, 2, MaxInt);
end;

class function TMyUtils.TitleCase(const S: string): string;
begin
  Result := TRegEx.Replace(S, '(\p{L}|\p{N})+', Callback)
end;

var
  S: string;
begin
  S := TMyUtils.TitleCase('hello world'); //Hello World
```

Advanced Unicode support

The basic multilingual plane and beyond

A single `Char` value can hold any character in the 'basic multilingual plane' (BMP). In most cases, this is more than enough, and in a worldwide setting too — European, Middle Eastern, African and Far Eastern languages are all covered, along with numerous symbols. However, some characters defined by the Unicode standard require two `Char` values, using what are called 'surrogate pairs'. Examples include certain musical and mathematical symbols, various Chinese characters (for example, the Han character 𠂢), and the symbols of many ancient languages.

For dealing with surrogates, the RTL defines a small set of helper routines:

- To test whether a given `Char` value is part of a surrogate pair, call either a relevant method of `TCharacter` (`IsSurrogate`, `IsLowSurrogate`, `IsHighSurrogate`, `IsSurrogatePair`) or the `IsLeadChar` function of `System.SysUtils`. Note that the *first* element in a surrogate pair is the 'high' surrogate, and the second the 'low' one:

```
uses System.Character;

const
  Test = '𠂢';
begin
  WriteLn(IsLowSurrogate(Test[1]));  //output: FALSE
  WriteLn(IsHighSurrogate(Test[1])); //output: TRUE
  WriteLn(IsLowSurrogate(Test[2]));  //output: TRUE
  WriteLn(IsHighSurrogate(Test[2])); //output: FALSE
end.
```

  For a `Char`/`WideChar` value, `IsLeadChar` does exactly the same thing as `TCharacter.IsSurrogate`.

- To define in code a character outside the BMP, use either a string literal, a pair of `Char` numerical literals, or the `ConvertFromUtf32` method of `TCharacter`, which takes a Unicode ordinal value and outputs a string containing the character concerned. The following example demonstrates each of the three approaches, all with respect to the same character:

```
uses System.Character, Vcl.Dialogs;

begin
  ShowMessage('𠂢');
  ShowMessage(#$D840#$DC8A);
  ShowMessage(TCharacter.ConvertFromUtf32($2008A));
end.
```

- A surrogate aware version of `Length` is `ElementToCharLen`, a function declared in `System.SysUtils`. This takes two parameters, one for the source string and one for the maximum number of `Char` values to parse. In the case of the second, just pass `MaxInt` to always parse to the end of the string. `ElementToCharLen('𠂢 ', MaxInt)` therefore returns 1 where `Length('𠂢 ')` returns 2.
- To calculate the number of `Char` values taken up by the first *n* characters of a string, call `CharToElementLen`. For example, `CharToElementLen('The 𠂢 symbol', 8)` returns 9, i.e. one more than the number of characters counted given there is one surrogate pair in the string.
- To find the `Char` index of a given character index, call `ElementToCharIndex`; to go the other way, call `CharToElementIndex`.

Decomposed characters

Unfortunately, surrogates are not the only way a string could hypothetically require more than one `Char` value per character. The second are 'decomposed' characters, which are when a letter that has one or more diacritical marks is encoded in separate `Char` values, one for the basic letter and one for each diacritic. For example, the word `café` may either be encoded as four `Char` values `c`, `a`, `f` and `é`, or as five — `c`, `a`, `f`, `e` and `#$0301` (`#769` in decimal), i.e. `cafe` followed by a decomposed accent.

Decomposed characters are pretty rare, which is just as well because most Delphi RTL functions do not take account of them. The main exceptions are the whole string comparison routines that have locale sensitivity, i.e. `SameStr`, `SameText`, `CompareStr` and `CompareText` when `loUserLocale` is passed, or `AnsiSameStr`, `AnsiSameText`, `AnsiCompareStr` and `AnsiCompareText`. All correctly understand composed and decomposed characters as equal. In contrast, the equality and inequality operators do not, along with functions such as `StartsStr`:

```
uses System.SysUtils, System.StrUtils;
```

```
const
  Precomposed = 'café';
  Decomposed = 'cafe'#$0301;

var
  OK: Boolean;
begin
  OK := (Precomposed = Decomposed);                    //FALSE
  OK := SameStr(Precomposed, Decomposed, loUserLocale); //TRUE
  OK := AnsiSameStr(Precomposed, Decomposed);          //TRUE
  OK := StartsText(Decomposed, 'CAFÉ CULTURE'));       //FALSE
end.
```

Worse, there are no RTL functions available to normalise strings into containing either just composed or decomposed characters — for that, you need to go down to the operating system level. On Windows, the `FoldString` API function can do the deed in either direction:

```
uses Winapi.Windows, System.SysUtils;

function EnsurePrecomposed(const S: string): string;
begin
  if S = '' then Exit('');
  SetLength(Result, Length(S));
  SetLength(Result, FoldString(MAP_PRECOMPOSED, PChar(S),
    Length(S), PChar(Result), Length(S)));
end;

function EnsureDecomposed(const S: string): string;
begin
  if S = '' then Exit('');
  SetLength(Result, FoldString(MAP_COMPOSITE, PChar(S),
    Length(S), nil, 0));
  if FoldString(MAP_COMPOSITE, PChar(S), Length(S),
    PChar(Result), Length(Result)) = 0 then RaiseLastOSError;
end;

const
  PrecomposedStr = 'café';
  DecomposedStr = 'cafe'#$0301;
var
  S: string;
  OK: Boolean;
begin
  OK := (PrecomposedStr = DecomposedStr); //FALSE
  S := EnsurePrecomposed(DecomposedStr);
  OK := (PrecomposedStr = S);             //TRUE
  S := EnsureDecomposed(PrecomposedStr);
  OK := (S = DecomposedStr);              //TRUE
end.
```

On OS X, the `CFStringNormalize` function (part of the 'Core Foundation' API) does a very similar job to `FoldString` on Windows. To use it, you need to first convert the source string to a Core Foundation `CFMutableString`, call `CFStringNormalize` itself, before converting back to a Delphi string:

```
uses Macapi.CoreFoundation;

function NormalizeViaCF(const S: string;
  NormForm: CFStringNormalizationForm): string;
var
  Range: CFRange;
  Ref: Pointer;
begin
  if S = '' then Exit('');
  //create a CFMutableString and copy over the chars
  Ref := CFStringCreateMutable(nil, 0);
  CFStringAppendCharacters(Ref, PChar(S), Length(S));
  //normalise the copied characters
  CFStringNormalize(Ref, NormForm);
  //copy over the chars back over to a Delphi string
  Range.location := 0;
  Range.length := CFStringGetLength(Ref);
  SetLength(Result, Range.length);
  CFStringGetCharacters(Ref, Range, PChar(Result));
end;

function EnsurePrecomposed(const S: string): string;
begin
```

```
  Result := NormalizeViaCF(S, kCFStringNormalizationFormC);
end;

function EnsureDecomposed(const S: string): string;
begin
  Result := NormalizeViaCF(S, kCFStringNormalizationFormD);
end;
```

Unicode encodings beyond UTF-16

In using UTF-16, or more exactly, UTF-16 LE ('little endian'), Delphi strings employ the same encoding scheme used by the Windows API, Apple's Cocoa frameworks on OS X, .NET, Java, and more. This is not the only encoding the Unicode standard defines however, since there is also UTF-8, UTF-16 BE ('big endian'), UTF-32 LE and UTF-32 BE (UTF-32 is sometimes called 'UCS-4' — Universal Character Set, 4 bytes).

In UTF-8, a human-readable character can take anything between 1 and 4 bytes; in contrast, UTF-16 BE is like UTF-16 LE but with the byte order of each element reversed. For example, where the Euro sign (€) has an ordinal value of 8364 ($20AC in hexadecimal notation) using UTF-16 LE, it is 44064 ($AC20) using UTF-16 BE. As for the two forms of UTF-32, they use four bytes per element. These four bytes come at the cost of needing extra memory compared to UTF-16, but remove the need for surrogate pairs. Alas, but the possibility of decomposed characters stands for UTF-32 as much as it does for UTF-16 (or UTF-8), so the absence of surrogates in UTF-32 doesn't actually mean you can always assume one element = one human readable character even there. In practice, this means UTF-32 is rarely used, and Delphi's support for it is correspondingly slight. Nevertheless, support does exist.

UTF-32/UCS-4 support (UCS4Char, UCS4String, TCharacter)

The `System` unit provides the following routines:

```
type
  UCS4Char = type LongWord;
  UCS4String = array of UCS4Char;

function UnicodeStringToUCS4String(
  const S: UnicodeString): UCS4String;
function UCS4StringToUnicodeString(
  const S: UCS4String): UnicodeString;
function PUCS4Chars(const S: UCS4String): PUCS4Char;
```

The `TCharacter` type then adds the following class methods:

```
class function ConvertFromUtf32(C: UCS4Char): string;
class function ConvertToUtf32(const S: string;
  Index: Integer): UCS4Char;
class function ConvertToUtf32(const S: string;
  Index: Integer; out CharLength: Integer): UCS4Char;
class function ConvertToUtf32(const HighSurrogate,
  LowSurrogate: Char): UCS4Char; overload;
```

These define a UTF-32/UCS-4 string as a dynamic array of 32 bit unsigned integers. Unlike a genuinely built-in string type, `UCS4String` therefore indexes from 0 rather than 1. The functions then work out as thus:

- `UnicodeStringToUCS4String` converts a regular, UTF-16 string to a UTF-32 one, with `UCS4StringToUnicodeString` converting in the opposite direction.
- `TCharacter.ConvertToItf32` and `TCharacter.ConvertFromUtf32` do the same thing for individual characters.
- Lastly, `PUCS4Chars` is a simple function that mimics what happens when you cast a normal string to a `PChar`: if the source string is empty (and therefore, `nil`), a pointer to a null terminator is returned, otherwise a pointer to the string's data is.

Collectively, these routines amount to no more than support for UTF-32 as an interchange format. Thus, there is no support for assigning string literals to `UCS4String`, for example, or character literals to a `UCS4Char` — from the compiler's point of view, instances of these types are just standard dynamic arrays and unsigned integers respectively.

UTF-8 support

UTF-8 is a popular Unicode encoding for saved or persisted text, primarily due to it being a strict superset of ASCII and, as a result, relatively space efficient. While the number of bytes per character varies, only one byte is needed for unadorned letters like 'a' or 'Z', along with basic punctuation symbols like full stops and commas. Compared to UTF-16, this makes UTF-8 a much more space efficient format when predominantly English text is being encoded. Indeed, UTF-16

doesn't even regain the advantage for text that is predominantly Greek, Cyrillic or Arabic (UTF-8 is about equal due to its clever encoding scheme). Sometimes it is also argued that UTF-8 is 'compatible' with old, English-centric text editors. That is pushing it though — when resaving a UTF-8 encoded file, such an editor will corrupt both accented letters and common symbols (e.g. the curled or 'smart' quote characters ", ", ‘ and ’), let alone any characters not found in English.

As we have already seen, Delphi's basic support for UTF-8 comes in the form of the `UTF8String` type, which enables easy conversions to and from native UTF-16 strings using typecasts. In general, the Delphi RTL does not provide functions to manipulate `UTF8String` instances directly however — you can index them and get their length in bytes, but `CompareText` and the like are not overloaded to take them as well as native strings. Instead, the idea is to use `UTF8String` as an interchange type, getting the data into a native string as soon as possible. Nonetheless, a few UTF-8 specific routines are still defined.

Detecting valid UTF-8

The first three UTF-8 specific routines concern detecting valid UTF-8. All three are declared by the `System.WideStrUtils` unit:

```
type
  TEncodeType = (etUSASCII, etUTF8, etANSI);

function HasExtendCharacter(const S: RawByteString): Boolean;
function DetectUTF8Encoding(const S: RawByteString): TEncodeType;
function IsUTF8String(const s: RawByteString): Boolean;
```

`HasExtendCharacter` does a simple scan of the source string and returns `True` if it contains at least one `AnsiChar` value outside the ASCII range, and `False` otherwise. `DetectUTF8Encoding`, in contrast, returns `etUSASCII` when the source string contains only ASCII characters, `etUTF8` if it contains valid UTF-8 sequences beyond the ASCII range, and `etANSI` when there is at least one non-ASCII character that *isn't* part of a valid UTF-8 sequence. Lastly, the slightly misnamed `IsUTF8String` is a simple wrapper round `DetectUTF8Encoding` that returns `True` on `etUTF8` and `False` otherwise. This is misleading to the extent a purely ASCII string is valid UTF-8. Consequently, a better wrapper function might look like this:

```
function IsValidUTF8(const S: RawByteString): Boolean;
begin
  Result := DetectUTF8Encoding(S) in [etUSASCII, etUTF8];
end;
```

Elements and characters

Aside from the detection functions, `System.WideStrUtils` also provides routines for determining what sort of thing the individual `AnsiChar` elements of a `UTF8String` represent: a 'lead' byte (i.e., first byte of two or more that collectively denote a single human-readable character), a 'trail' byte (i.e., the final byte of such a group), or neither:

```
function IsUTF8LeadByte(Lead: AnsiChar): Boolean;
function IsUTF8TrailByte(Lead: AnsiChar): Boolean;
function UTF8CharSize(Lead: AnsiChar): Integer;
function UTF8CharLength(Lead: AnsiChar): Integer;
```

Both `UTF8CharSize` and `UTF8CharLength` return the number of bytes a character beginning with the specified `AnsiChar` value should take. The difference between the two functions is what is returned when this value does not denote a lead byte: where `UTF8CharSize` returns 0, `UTF8CharLength` returns 1.

As such, there is no stock function provided that goes on to compute the number of characters encoded by a given `UTF8String`. While the `Length` standard function works, it returns the number of elements, which in the case of `UTF8String` will be the number of `AnsiChar` values, and therefore, the number of bytes. Nonetheless, it is not hard to write a custom function that returns the number of characters, leveraging `UTF8CharLength`:

```
function NumOfCharactersInUTF8(const S: UTF8String): Integer;
var
  SeekPtr, NullTermPtr: PAnsiChar;
begin
  Result := 0;
  if S = '' then Exit;
  SeekPtr := PAnsiChar(S);
  NullTermPtr := @SeekPtr[Length(S)]; //PAnsiChar is 0 indexed
  repeat
    Inc(Result);
    Inc(SeekPtr, UTF8CharLength(SeekPtr^));
  until SeekPtr >= NullTermPtr;
end;

var
```

```
  S: UTF8String;
begin
  S := 'cafe';
  WriteLn('No. of elements = ', Length(S),
    ', no. of characters = ', NumOfCharacters(S));
  S := 'café';
  WriteLn('No. of elements = ', Length(S),
    ', no. of characters = ', NumOfCharacters(S));
end.
```

The first line outputted by the test code in this example shows both an element and character count of 4. In contrast, the second shows an element count of 5 and a character count of 4 due to the accented 'e' taking two bytes.

Byte order marks

Because of 'endianness' issues, encoding text in either UTF-16 or UTF-32 can easily lead to ambiguity. The problem concerns the internal layout of integers larger than one byte: while some computer systems put the 'high byte' first ('big endian'), others lead with the 'low byte' ('little endian'). Since UTF-16 encodes characters in double byte ('word') numbers, and UTF-32 in four byte ('long word') ones, this results in big endian and little endian variants of each.

To distinguish between them, the Unicode standard suggests a small byte sequence or 'byte order mark' (BOM) be put at the head of a UTF-16 or UTF-32 encoded text file. In the case of UTF-16 BE, the BOM is $FE followed by $FF; for UTF-16 LE, $FF followed by $FE. In practice, UTF-16 BE is very rare, especially in a Delphi context due to the fact the x86/x64 processor architectures targeted by the Delphi compiler have always been little endian. Nonetheless, the possibility of UTF-16 BE encoded files still exists.

When you turn to UTF-8, in contrast, the idea of a BOM does not make much sense when taken literally. This is because UTF-8 is encoded in a way that does not allow for byte order ambiguity in the first place. Nonetheless, a common convention on Windows is to begin UTF-8 encoded files with their own 'BOM' — the byte sequence $EF, $BB and $BF — in order to easily distinguish them from text files written using a legacy 'Ansi' encoding.

For convenience, the `WideStrUtils` unit defines the UTF-8 BOM as a constant, `sUTF8BOMString`. It also has a couple of functions to quickly determine whether this mark is at the head of either a stream or any `AnsiString`/`UTF8String` variant:

```
function HasUTF8BOM(S: TStream): Boolean; overload;
function HasUTF8BOM(const S: RawByteString): Boolean; overload;
```

In the case of the stream version, it always checks the start of the stream, putting the `Position` property back to where it was before afterwards.

Converting between character encodings: the TEncoding class

Partly based on the concept of a BOM, the `TEncoding` class, declared in `System.SysUtils`, provides a standard means for converting between different text encodings. The implementation of `TEncoding` takes the form of an abstract class, with concrete descendants for common encodings declared as static class properties on `TEncoding` itself:

```
class property ASCII: TEncoding read GetASCII;
class property BigEndianUnicode: TEncoding
  read GetBigEndianUnicode;
class property Default: TEncoding read GetDefault;
class property Unicode: TEncoding read GetUnicode;
class property UTF7: TEncoding read GetUTF7;
class property UTF8: TEncoding read GetUTF8;
```

Here, 'Unicode' means UTF-16 LE, 'big endian Unicode' UTF-16 BE, 'default' the system legacy code page on Windows, and UTF-8 on OS X.

`TEncoding` instances for other code pages — which typically means Windows 'Ansi' code pages, but possibly old Mac OS ones too — can be created via the `GetEncoding` static function, which takes either a string or numerical identifier. For example, both `932` and `'shift_jis'` will instantiate a `TEncoding` descendant for the Windows 'Ansi' code page for Japan, underlying operating support permitting:

```
var
  Encoding: TEncoding;
begin
  Encoding := TEncoding.GetEncoding('shift_jis');
  //work with Encoding...
```

Be warned that while the `TEncoding` instances declared as class properties on `TEncoding` itself are 'owned' by the class, those returned by `GetEncoding` are not, leaving you responsible for their destruction

Performing conversions and determining input encodings

To covert between encodings, call the `Convert` class function of `TEncoding`:

```
class function Convert(Source, Destination: TEncoding;
  const Bytes: TBytes): TBytes;
class function Convert(Source, Destination: TEncoding;
  const Bytes: TBytes; StartIndex, Count: Integer): TBytes;
```

Here's it used to convert a stream assumed to hold UTF-7 text into a UTF-8 buffer:

```
function ConvertUTF7StreamToUTF8(Stream: TStream): TBytes;
var
  SourceBytes: TBytes;
begin
  Stream.Position := 0;
  SetLength(SourceBytes, Stream.Size);
  Stream.ReadBuffer(Result[0], Length(SourceBytes));
  Result := TEncoding.Convert(TEncoding.UTF7,
    TEncoding.UTF8, SourceBytes);
end;
```

Probably more useful than `Convert` is `GetString`. This is an instance not a class method, taking a `TBytes` buffer and decoding its contents into a string:

```
function GetString(const Bytes: TBytes): string; overload;
function GetString(const Bytes: TBytes;
  ByteIndex, ByteCount: Integer): string; overload;
```

For example, the following decodes a stream assumed to hold text encoded in the big endian variant of UTF-16 BE:

```
function ConvertUTF8StreamToString(Stream: TStream): string;
var
  Bytes: TBytes;
begin
  if Stream is TBytesStream then
    Bytes := TBytesStream(Stream).Bytes
  else
  begin
    Stream.Position := 0;
    SetLength(Bytes, Stream.Size);
    Stream.ReadBuffer(Bytes[0], Length(Bytes));
  end;
  Result := TEncoding.BigEndianUnicode.GetString(Bytes)
end;
```

To go the other way, call `GetBytes`:

```
var
  Bytes: TBytes;
begin
  Bytes := TEncoding.UTF8.GetBytes('Blah de blah');
```

Determining the code page to use

To determine which `TEncoding` instance to use in the first place, you can call the `GetBufferEncoding` class method:

```
class function GetBufferEncoding(const Buffer: TBytes;
  var AEncoding: TEncoding): Integer;
class function GetBufferEncoding(const Buffer: TBytes;
  var AEncoding: TEncoding; ADefault: TEncoding): Integer;
```

The first variant simply calls the second, passing `TEncoding.Default` for the final parameter. Whichever version you call, you should initialise `AEncoding` to `nil` — if you don't, then `GetBufferEncoding` will do nothing but check for whether the specified encoding's 'preamble' (i.e., BOM) is at the head of `Buffer`, returning the preamble's size if it is and zero otherwise.

In contrast, when `AEncoding` is `nil` on input, the BOMs for UTF-16 LE, UTF-16 BE and UTF-8 are looked for. If a match is found, then `AEncoding` is assigned, and the length of the found BOM returned as the function result. If nothing is matched, then `AEncoding` is set to `ADefault`, with the size of its BOM returned *if* that BOM is matched, otherwise the function returns zero. Whatever happens, `AEncoding` is assigned to something.

Unfortunately, `GetBufferEncoding` is not extensible — if you wish to allow detecting BOMs beyond the standard three, then you'll have to code it yourself. Moreover, the function makes no attempt to check whether the text really is encoded by what its BOM says it is, or if no BOM is matched, what the text *might* be encoded in.

C-style ‘strings’ (PChar/PWideChar and PAnsiChar)

‘C-style strings’ are pointers to character buffers that end in a #0 — a ‘null terminator’. Typewise, they are represented in Delphi by `PAnsiChar`, `PWideChar` and `PChar`: `PAnsiChar` points to a buffer of single byte `AnsiChar` elements, and both `PWideChar` and `PChar` point to buffers of double byte `WideChar` elements. `PWideChar` and `PChar` are equivalent; in principle, `PChar` could map onto something else in the future, though this is unlikely (in the past it mapped to `PAnsiChar`).

As such, a C-style string is not really a true string. This is because it implies nothing about how the memory it points to was allocated — it is just a pointer. This means you cannot dynamically reallocate a `PChar` in the way you can dynamically reallocate a `string` (say, by concatenating one string to another), since unless you did the original allocation yourself, you simply do not have the knowledge to safely *re*allocate it, or even deallocate it. With a proper string this problem doesn’t arise, since a string’s memory is managed for you by the compiler and RTL.

As with pointers generally, use of C-style strings isn’t typical in normal Delphi programming. Nonetheless, they do serve two purposes in particular: for interoperability with operating system APIs and other C-compatible interfaces, and for writing optimised string handling routines. In the first case they allow easily converting a Delphi string to and from the raw character pointers used by the Windows and POSIX APIs; in the second, they allow scanning over string data in a slightly more performant way than normal string indexing. In neither case are they usually employed *in place* of native strings however — rather, they supplement them, or even just give a different ‘view’ of their data.

Converting from a string to a PChar

Technically, a Delphi string *is* a `PChar`. This is because the low-level RTL always appends an implicit null terminator to the end of a string’s normal character data. Thus, if a string is assigned `'abc'`, then under the bonnet it actually gets given the characters `'abc'#0` to point to.

The reason for this is to allow easy ‘conversions’ from a `string` to a `PChar`: simply use a typecast, either to `PChar` itself or `Pointer`:

```
var
  S: string;
  P: PChar;
begin
  S := 'Hello C-style string world';
  P := PChar(S);
  P := Pointer(S);
```

The difference between the two only arises when the string being assigned is empty (i.e., equals `''`). If so, a `Pointer` cast will return `nil` where a `PChar` one will return a pointer to a null terminator:

```
var
  S: string;
  P: PChar;
begin
  S := '';                        //assign an empty string
  P := PChar(S);
  WriteLn('Is P nil? ', P = nil); //output: FALSE
  P := Pointer(S);
  WriteLn('Is P nil? ', P = nil); //output: TRUE
```

Generally, the `PChar` behaviour is what you want. This is because it makes life easier for the consuming code, relieving it of the need to check the pointer given isn’t `nil` before dereferencing.

Converting from a PChar to a string

Converting from a `PChar` to a `string` is even simpler than converting from a `string` to a `PChar`: just make the assignment, no typecast required:

```
var
  S1, S2: string;
  P: PChar;
begin
  S1 := 'This is some text';
  P := PChar(S1);
  S2 := P;
  WriteLn(S2); //output: This is some text
```

Whereas going from a `string` to a `PChar` doesn’t copy any data, going from a `PChar` to a `string` does, despite the simpler syntax. Moreover, this copying need not be superfast, since in order to copy characters the low-level RTL needs to know how many there are, which it can only learn by traversing the buffer pointed to the `PChar` until it finds the null terminator.

Because of this, if you already know how many characters the `PChar` points to, you should avoid a simple assignment and use the `SetString` standard procedure instead:

```
var
  S1, S2: string;
  P: PChar;
  PLen: Integer;
begin
  S1 := 'This is some more text';
  P := PChar(S1);
  PLen := Length(S1);
  SetString(S2, P, PLen);
  WriteLn(S2); //output: This is some more text
```

A popular use of `SetString` is when writing wrappers for Windows API functions like `GetWindowsDirectory`. These routines take a character buffer of `MAX_PATH` elements long (`MAX_PATH` is a special constant defined by the API), before filling the buffer with the requested data and returning the number of characters actually copied, minus the null terminator, as their function result.

To make calling this sort of thing easier, `PChar` is type assignable with zero-indexed static arrays of `Char` (similarly, `PAnsiChar` is type assignable with zero-indexed static arrays of `AnsiChar`). A simple wrapper function for `GetWindowsDirectory` can therefore be written like this:

```
uses Winapi.Windows;

function GetWindowsDir: string;
var
  Buffer: array[0..MAX_PATH] of Char;
begin
  SetString(Result, Buffer,
    GetWindowsDirectory(Buffer, Length(Buffer)));
end;

begin
  WriteLn(GetWindowsDir);
end.
```

This outputs `C:\Windows` on my computer.

Dealing with 'Ansi' and UTF-8 encoded C-style strings

As a `string` can be safely typecast to a `PChar`, so an `AnsiString` can be safely typecast to a `PAnsiChar`. Similarly, a `PAnsiChar` is directly assignable to an `AnsiString`, and `SetString` can be used between the two as well:

```
var
  S1, S2, S3: AnsiString;
  P: PAnsiChar;
begin
  S1 := 'Hello "Ansi" world!';
  P := PAnsiChar(S1);
  WriteLn(P);                  //output: Hello "Ansi" world!
  S2 := P;
  WriteLn(S2);                 //output: Hello "Ansi" world!
  SetString(S3, P, Length(S1));
  WriteLn(S3);                 //output: Hello "Ansi" world!
```

Since a `UTFString` is internally implemented as a special sort of `AnsiString`, the same goes for it too. An important caveat however is that by itself, a `PAnsiChar` holds no information about the encoding used, in contrast to an `AnsiString` or `UTF8String`, which 'knows' it holds (say) Latin-1 or UTF-8 data. This means that if a C API requires `PAnsiChar` input, you have to know independently what encoding it expects. In many cases this will be apparent either from the function names or the documentation, but you will have check — the compiler won't be able to help you decide, or hint that you may have got it wrong.

In contrast, you will get a compiler error if you attempt to pass a `PChar` to a routine expecting a `PAnsiChar` or vice versa:

```
procedure Foo(Buffer: PAnsiChar);
begin
end;

var
  S: string;
begin
  S := 'Test';
  Foo(PChar(S)); //compiler error: incompatible types
```

While you might expect casting directly from a `string` to a `PAnsiChar` would be an error too, it actually only produces a warning ('suspicious typecast'. The fix is to make an intervening cast to an `AnsiString` or `UTF8String` as appropriate:

```
procedure Foo2(UTF8Data: PAnsiChar);
begin
end;

var
  S: string;
begin
  S := 'Test';
  Foo(PAnsiChar(S)); //compiler warning: suspicious typecast
  Foo(PAnsiChar(UTF8String(S))); //compiles cleanly
```

Without the intervening cast, you aren't actually converting anything — instead, a pointer to the string's original UTF-16 data is returned.

Working with PChar variables

Being at root a pointer to a `Char`, dereferencing a `PChar` with the `^` symbol returns the first character pointed to:

```
var
  S: string;
  P: PChar;
begin
  S := 'ABCDEF';
  P := PChar(S);
  WriteLn(P^);   //output: A
```

If you *don't* deference a `PChar` when assigning or passing it to something that expects a string, an automatic conversion will kick in:

```
  WriteLn(P);   //output: ABCDEF
```

A standard feature of typed pointers is that they work with the `Inc` and `Dec` standard routines. When called, these will increment and decrement the pointer by the size of the thing pointed to. In the `PChar`/`PWideChar` case, this moves the pointer on by two bytes, given `SizeOf(WideChar) = 2`. So, when `P` is a `PChar` and `Inc(P, 2)` is called, the pointer is moved on by two characters, not two bytes.

The effect of calling `Inc` or `Dec` on a `PChar` or `PAnsiChar` is to have it 'forget' that its data 'really' starts somewhere other than where it is now pointing to:

```
var
  S: string;
  P: PChar;
begin
  S := 'Testing';
  P := PChar(S);
  Inc(P);
  WriteLn('P now points to the letter ', P^);    //P^ = e
  WriteLn('As a "string", P now points to ', P); //P = esting
  Inc(P, 2);
  WriteLn('P now points to the letter ', P^);    //P^ = t
  WriteLn('As a "string", P now points to ', P); //P = ting
```

It is perfectly safe to do this, so long as you remember a `PChar` (or `PAnsiChar`) is *just a pointer* — if you increment beyond the end of the source data and attempt to read off the pointer afterwards, your best hope is for the program to crash immediately, otherwise you may end up corrupting memory in hard-to-debug ways.

Aside from simply incrementing a `PChar`, you can also write to it:

```
function ReplaceLetter(const S: string;
  OldLetter, NewLetter: Char): string;
var
  SeekPtr: PChar;
begin
  SetString(Result, PChar(S), Length(S));
  SeekPtr := PChar(Result);
  while SeekPtr^ <> #0 do
  begin
    if SeekPtr^ = OldLetter then SeekPtr^ := NewLetter;
    Inc(SeekPtr);
  end;
end;

var
```

```
  S: string;
  P: PChar;
begin
  S := ReplaceLetter('Good? Great?', '?', '!'); //Good! Great!
```

Notice this example calls `SetString` rather than performing a simple assignment of `S` to `Result`. This is to ensure the string being worked on points to its own data. Normally you don't have to worry about this due to Delphi strings' 'copy on write' behaviour, however editing a string through a pointer subverts that. An alternative to calling `SetLength` is to make the assignment, but call `UniqueString` immediately afterwards:

```
Result := S;
UniqueString(Result);
```

PChar indexing

Notwithstanding how the `SetLength` call makes our `ReplaceLetter` example safe with respect to string reference counting, it is still arguably flawed overall though. This is because it assumes the first null terminator it encounters is the closing, implicit null terminator of the string. In principle, Delphi strings can contain 'embedded nulls' however, i.e. #0 characters as part of its regular data. For example, if you called

```
S := ReplaceLetter('Naughty?'#0'String?', '?', '!');
```

then `S` will be set to `'Naughty!'#0'String?'`. Whether this limitation actually matters depends though. In particular, it is generally bad form to allow embedded nulls in strings intended for display, since underlying operating APIs, being C based, are likely not to understand them.

Nonetheless, our `ReplaceLetter` function could be easily enough fixed by using `PChar` indexing:

```
function ReplaceLetter(const S: string;
  OldLetter, NewLetter: Char): string;
var
  P: PChar;
  I: Integer;
begin
  SetString(Result, PChar(S), Length(S));
  P := PChar(Result);
  for I := 0 to Length(S) - 1 do
    if P[I] = OldLetter then P[I] := NewLetter;
end;
```

This produces slightly more optimised code compared to an equivalent function that uses native string indexing. However, watch out for the fact a `PChar` (or `PAnsiChar`) indexes from 0, not 1. Furthermore, where string indexing can be made 'safe' by enabling range checking, such a thing doesn't exist (or even make sense) in the land of C-style strings.

However, correlative to the fact you can't enable range checking on a `PChar` is the fact negative indexes are formally valid:

```
var
  S: string;
  P: PChar;
begin
  S := 'ABCDEF';
  P := PChar(S);
  Inc(P, 4);
  WriteLn(P^);    //E
  WriteLn(P[-2]); //C
```

When using negative indexing, you should be particularly careful to ensure you will be indexing to valid memory.

A final little trick up `PChar`'s sleeve is an ability to subtract one instance from another. This returns the number of characters between the two pointers:

```
function NumOfCharsBetweenXs(const S: string): Integer;
var
  Start, Finish: PChar;
begin
  Start := PChar(S);
  repeat
    if Start^ = #0 then Exit(-1);
    if Start^ = 'X' then Break;
    Inc(Start);
  until False;
  Finish := Start;
  repeat
    Inc(Finish);
    if Finish^ = #0 then Exit(-1);
```

```
  until (Finish^ = 'X');
  Result := Finish - Start - 1; //- 1 to make it exclusive
end;

var
  Num: Integer;
begin
  Num := NumOfCharsBetweenXs('.X........X.'); //8
```

In this example, the two `PChar` values point to somewhere within the same character buffer or block of memory. What if you compare two `PChar` values that point to completely separate buffers however? While you won't cause a crash, you won't get a meaningful result either.

6. Arrays, collections and enumerators

This chapter looks in depth at types for grouping pieces of data together. It begins with arrays, a basic yet not always properly understood language feature, before turning to provide an in-depth guide to using the collection classes provided by the RTL (lists, 'dictionaries', and so forth). In its final section, we will then look at writing linked lists and custom 'enumerators', relatively advanced topics that concern the foundations of implementing your own collection classes.

Arrays

An 'array' is a variable that holds a sequence of values, all of the same type, that are accessed by an index. Arrays in Delphi come in two main forms, 'static' and 'dynamic'. Static arrays are so called because they are allocated on the stack, and because of that, have dimensions that are fixed at compile time. Dynamic arrays, in contrast, are allocated on the heap, and have dimensions that are set 'dynamically' — i.e., at runtime — by calling the `SetLength` standard procedure.

The syntax for reading and writing array elements is consistent regardless of the sort of array used — just type the identifier of the array followed by the index in square brackets:

```
I := MyArray[3]; //read off the element at index 3
MyArray[3] := J; //set the element at index 3
```

The syntax for declaring an array in the first place can vary though. For example, the following code declares single dimensional string arrays in various ways:

```
//declare explicit array types
type
  TMyStaticArray = array['a'..'z'] of string;
  TMyDynStringArray = array of string;
  TRodent = (Mouse, GuineaPig, Rat);

//declare an array typed constant
const
  StaticArrConst: array[1..2] of string = ('Hello', 'World');

var
  StaticArr1: array[0..9] of string; //inline static array type
  DynArr1: array of string;          //inline dynamic array type
  StaticArr2: array[Byte] of string; //equiv. to array[0..255]
  DynArr2: TArray<string>;           //dynamic string array
  StaticArr3: TMyStaticArray;        //use type declared above
  DynArr3: TMyDynStringArray;        //ditto
  StaticArr4: array[TRodent] of string;
begin
  StaticArr1[0] := 'Simple';
  SetLength(DynArr1, 2);
  DynArr[0] := 'examples';
  StaticArr2[0] := 'can';
  DynArr2 := TArray<string>.Create('nevertheless', 'be');
  StaticArr3['a'] := 'quite';
  DynArr3 := TMyDynStringArray.Create('instructive', 'I');
  StaticArray[Mouse] := 'think';
```

The potential for complexity has three main causes:

- While dynamic arrays are always indexed with an integer that starts at 0 for the first item, static arrays support using *any* ordinal type for their indexer. The code above illustrates this with an array that uses an enumerated type (`StaticArr4`) and another that uses a `Char` sub-range (`StaticArr3`).
- Custom array types can be declared (`TMyStaticArray` and `TMyDynStringArray`) and used (`StaticArr3`, `DunArr3`).
- Even without custom types, dynamic arrays can be declared either 'raw' (`DynArr1`) or by instantiating `TArray`, a predefined generic type (`DynArr2`).
- Using some form of explicit type allows initialising a dynamic array using a 'constructor' syntax (`DynArr2`, `DynArr3`) which otherwise is not available (`DynArr1`). Nonetheless, the `SetLength` standard procedure works regardless of whether an explicit dynamic array type is used or not.

Notwithstanding the complications involved, these reasons are generally nice ones. For example, the fact a static array can be indexed to an enumeration means you can map the elements of an enumerated type to strings very easily:

```
const
  RodentStrings: array[TRodent] of string = ('Mouse',
    'Guinea pig', 'Rat');

var
  Rodent: TRodent;
begin
  Rodent := GuineaPig;
  WriteLn(RodentStrings[Rodent]); //output: Guinea pig
```

The constructor syntax offered by explicit dynamic array types is also quite handy, though it is no more than a shortcut. For example, this:

```
var
  DynArr: TArray<string>;
begin
  DynArr := TArray<string>.Create('First', 'Second', 'Third', 'Fourth');
```

maps to this:

```
var
  DynArr: TArray<string>;
begin
  SetLength(DynArr, 4);
  DynArr[0] := 'First';
  DynArr[1] := 'Second';
  DynArr[2] := 'Third';
  DynArr[3] := 'Fourth';
```

Lastly, the ability to declare explicit types allows sharing arrays of the same form between different units. There is however a dark side to this: Delphi's arguably over-zealous array type compatibility rules.

Array type compatibility

At first glance, declaring explicit array types may seem a bit pointless. However, consider the following code:

```
var
  StaticArr1: array[0..1] of string;
  StaticArr2: array[0..1] of string;
begin
  StaticArr1[0] := 'Hello';
  StaticArr1[1] := 'World';
  StaticArr2 := StaticArr1;
```

Try to compile this, and you won't be able to for the sin of using 'incompatible types'. This issue affects dynamic arrays as well:

```
var
  DynArr1: array of string;
  DynArr2: array of string;
begin
  SetLength(DynArr1, 1);
  DynArr1[0] := 'Gah! Compiler error next line!';
  DynArr2 := DynArr1;
```

The reason is that two array variables must share the same type for one to be assigned to the other, and in these cases, the two variables in each pair have their own, implicit type. The fact the types concerned resolve to the same data structure does not matter!

In the case of local variables, you can fix this by making them share the same inline type:

```
var
  DynArr1, DynArr2: array of string; //now share their type
begin
  SetLength(DynArr1, 1);
  DynArr1[0] := 'Goodbye Compiler Error!';
  DynArr2 := DynArr1;
```

This solution cannot be applied to variables declared in different scopes though, e.g. to two arrays declared in different units. In that case, the variables concerned need to share an explicit type:

```
type
  TMyArray = array[0..1] of string;

var
  StaticArr1: TMyArray;
  StaticArr2: TMyArray;
begin
  StaticArr1[0] := 'This';
  StaticArr1[1] := 'compiles';
  StaticArr2 := StaticArr1;
```

This is the sole reason the `TArray` type we met earlier exists:

```
var
  DynArr1: TArray<string>;
  DynArr2: TArray<string>;
begin
  SetLength(DynArr1, 1);
  DynArr1[0] := 'Goodbye Compiler Error!';
  DynArr2 := DynArr1;
```

Copying static to dynamic arrays and vice versa

Unfortunately, the way array type compatibility is so strict means it is impossible to assign a static to a dynamic array or vice versa in one go: instead, elements must be copied over one at a time.

Nonetheless, if the element type is not a managed one (so, not `string` or so forth), you can use the `Move` standard procedure to do a block copy. Doing this requires care, since `Move` itself is not an array-specific routine, but a low-level procedure that simply copies the specified number of bytes from one variable to another. In the following, it is used to perform a block copy of a static array to a dynamic one. The `SizeOf` standard function used to retrieve the number of bytes taken by the static array:

```
const
  BigIntsToStart: array[1..3] of Integer = (777777, 888888, 999999);

function CreateFromBigInts: TArray<Integer>;
begin
  SetLength(Result, Length(BigIntsToStart));
  Move(BigIntsToStart, Result[0], SizeOf(BigIntsToStart));
end;
```

The dynamic array *must* be indexed when passed to `Move`, since in having a reference type, the variable merely points to its data. Whether you index or not in the static array case is up to you however. If you do, you must be careful to ensure you index the first element, which is easy to mess up given static arrays are not necessarily zero-indexed.

Here's an example of going the other way:

```
var
  Ints: array[1..3] of Integer;

procedure CopyToStaticInts(const DynArr: TArray<Integer>);
begin
  Assert(Length(DynArr) = Length(Ints));
  Move(DynArr[0], Ints, SizeOf(Ints));
end;
```

Notice that `SizeOf` is called against the static array again. If used on the dynamic array, it would always return 4 (if targeting Win32 or OS X) or 8 (if targeting Win64), i.e. the size of a pointer. To get the total size of the data referred to, you would need to call `SizeOf` on an *element* and multiply the result by the length of the array:

```
Move(DynArr[0], Ints, SizeOf(DynArr[0]) * Length(DynArr));
```

The same procedure actually works for a static array too, even if the indirection is unnecessary in that case:

```
WriteLn(SizeOf(Ints));                   //output: 12
WriteLn(SizeOf(Ints[1]) * Length(Ints)); //output: 12
```

Static vs. dynamic array semantics

Static arrays are value types where dynamic arrays are reference types. Assigning one static array to another therefore copies the values of all the items, whereas assigning one dynamic array to another merely copies a reference to the first array's values:

```
procedure ArrayAssignmentTest;
var
  StaticArray1, StaticArray2: array[0..1] of Integer;
  DynArray1, DynArray2: array of Integer;
begin
  { test static array assignment }
  StaticArray1[0] := 111;
  StaticArray1[1] := 222;
  StaticArray2 := StaticArray1;      //does a 'deep' copy
  StaticArray2[0] := 333;
  Writeln(StaticArray1[0]);          //prints 111
  Writeln(StaticArray2[0]);          //prints 333
  { test dynamic array assignment }
  SetLength(DynArray1, 2);
  DynArray1[0] := 111;
  DynArray1[1] := 222;
  DynArray2 := DynArray1;            //copies only a reference!
  DynArray2[0] := 333;
  WriteLn(DynArray1[0]);             //prints 333
  WriteLn(DynArray2[0]);             //prints 333
end;
```

If you want to replicate the static array behaviour with a dynamic array, you need to call the `Copy` standard function:

```
procedure ArrayAssignmentTest2;
var
  DynArray1, DynArray2: array of Integer;
begin
  SetLength(DynArray1, 2);
  DynArray1[0] := 111;
  DynArray1[1] := 222;
  DynArray2 := Copy(DynArray1);      //copy the data pointed to
  DynArray2[0] := 333;
  WriteLn(DynArray1[0]);             //prints 111
  WriteLn(DynArray2[0]);             //prints 333
end;
```

Another difference between static and dynamic arrays, again steaming from the value type/reference type distinction, is the fact a dynamic array can be assigned `nil`, and indeed be compared to `nil`. Assigning `nil` has the same effect as calling `SetLength` and passing 0 for the new length; similarly, when a dynamic array has a length of 0 elements, it is equal to `nil`.

Multidimensional arrays

Both static and dynamic arrays may be multidimensional, though only multidimensional dynamic arrays can be 'ragged' or non-rectangular:

```
procedure ExampleMultiDimArrays(Selector1, Selector2: Boolean);
const
  ArrConst: array[Boolean, Boolean] of Byte = ((0, 1), (2, 3));
var
  ChosenByte: Byte;
  StaticArr1: array[0..1, 0..9] of Integer;
  StaticArr2: array[0..1] of array[0..9] of Integer;
  DynArr1, DynArr2, DynArr3: array of array of Byte;
begin
  ChosenByte := ArrConst[Selector1, Selector2];
  //initialise DynArr1 in one go
  SetLength(DynArr1, 2, 10);
  //initialise DynArr2 dim by dim, but produce same result
  SetLength(DynArr2, 2);
  SetLength(DynArr2[0], 10);
  SetLength(DynArr2[1], 10);
  //make a ragged multi-dimensional array out of DynArr3
  SetLength(DynArr3, 2);
  SetLength(DynArr3[0], 5);
  SetLength(DynArr3[1], 15);
end;
```

For static arrays, the forms `array[x..y, a..b]` and `array[x..y] of array[a..b]` are equivalent. Using the first brings visual consistency with the way an element of a multidimensional array is read or set (`MyArr[X, Y] := Z`); using the second brings visual consistency with the way dynamic multidimensional arrays are declared, which is always in the form `array of array`.

As with their single dimensional cousins, multidimensional dynamic arrays have their size set by calling `SetLength`. If creating a ragged multidimensional array, `SetLength` must be called repeatedly, as with `DynArr3` above. In the case of a rectangular multidimensional array however, a single call can be used to set all dimensions at once, as with `DynArr1` in the example.

Range checking

By default, range checking is disabled for arrays, even in the standard Debug build configuration. This means if you attempt to index an array with a value that is out of bounds, it won't get caught unless it can be caught at compile time:

```
var
  NumsPerGrade: array['A'..'D'] of Integer;
  Grade: Char;
begin
  Grade := 'E';
  NumsPerGrade[Grade] := 999; //compiles & seems to run OK...
end.
```

At least for debug builds, it is worth enabling range checking to catch these errors when they happen. In the IDE, this can be set via the Build Configurations dialog; alternatively, in the actual source use `{$RANGECHECKS ON}` or `{$R+}`. To explicitly disable range checking, which can occasionally be useful in performance-sensitive code blocks, use `{$RANGECHECKS OFF}` or `{$R-}`. This works the same way it does for integers (cf. chapter 2).

Enumerating arrays

One way to enumerate an array is to use a `for`/`to` loop and explicit indexing. To get the lower and upper bounds, use the `Low` and `High` standard functions:

```
var
  Arr: array['A'..'Z'] of Integer;
  Ch: Char;
begin
  for Ch := Low(Arr) to High(Arr) do
    Arr[Ch] := Ord(Ch);
```

You can also call the `Length` standard function to return the number of elements. Since dynamic arrays are always zero-indexed, this will equal the result of `High` plus 1 in their case:

```
procedure SetStringArray(const Arr: TArray<string>);
var
  I: Integer;
begin
  for I := 0 to Length(Arr) - 1 do
    Arr[I] := IntToStr(I);
```

Arrays also support `for`/`in` loops. When used, these enumerate from the element with the lowest index to the element with the highest index. In the case of a multidimensional array, you can either specify which dimension to enumerate, or just enumerate the whole array, each dimension in turn:

```
var
  I, J, Total: Integer;
  TimesTable: array[1..5, 1..5] of Integer;
begin
  for I := 1 to 5 do
    for J := 1 to 5 do
      TimesTable[I, J] := I * J;
  Writeln('4 times table only:');
  for I in TimesTable[4] do
    WriteLn(' ', I);
  Writeln;
  Writeln('Everything:');
  for I in TimesTable do
    WriteLn(' ', I);
end.
```

Open arrays

Alongside static and dynamic arrays stands is a third sort: open arrays. These can be used in parameter lists only, and allow passing both static and dynamic array source values. As declared, they take on a syntax that looks the same as the 'raw' way to declare a dynamic array:

```
procedure OpenArrayTest(Ints: array of Integer);
var
  I: Integer;
begin
  for I in Ints do
    Write(I, ' ');
  WriteLn;
end;

var
  DynArr: TArray<Integer>;
  StaticArr: array[0..2] of Integer;
begin
  DynArr := TArray<Integer>.Create(1, 2, 3);
  OpenArrayTest(DynArr);    //output: 1 2 3
  StaticArr[0] := 11;
  StaticArr[1] := 22;
  StaticArr[2] := 33;
  OpenArrayTest(StaticArr); //output: 11 12 33
```

Inside the routine, an open array is always indexed like a dynamic array (i.e., using an integral indexer that starts from 0), even if the source is a static array that has a special indexer:

```
procedure OutputInfo(const Arr: array of string);
var
  I: Integer;
begin
  WriteLn('Low(Arr) = ', Low(Arr), '; High(Arr) = ', High(Arr));
```

```
  for I := Low(Arr) to High(Arr) do
    Write(' ' + Arr[I]);
  WriteLn;
end;

var
  StaticArr: array['y'..'z'] of string;
  DynArr: TArray<string>;
begin
  StaticArr['y'] := 'Static array source';
  StaticArr['z'] := 'to open array test';
  OutputInfo(StaticArr);
  DynArr := TArray<string>.Create(
    'Dynamic array source',
    'to open array test');
  OutputInfo(DynArr);
end.
```

This program outputs the following:

```
Low(Arr) = 0; High(Arr) = 1
 Static array source to open array test
Low(Arr) = 0; High(Arr) = 1
 Dynamic array source to open array test
```

In the case of a static array, you can pass just the first *n* elements to a routine that takes an open array by using the `Slice` standard function:

```
var
  StaticArr: array[1..4] of string;
begin
  StaticArr[1] := 'First';
  StaticArr[2] := 'Second';
  StaticArr[3] := 'Third';
  StaticArr[4] := 'Fourth';
  OutputInfo(Slice(StaticArr, 2));
```

This now outputs

```
Low(Arr) = 0; High(Arr) = 1
 First Second
```

The open array syntax is flexible enough to support `const` and `var` parameter modifiers. If you don't need to modify elements of the array passed in you should definitely use `const`, since that will avoid the source array being copied.

Given an open array must be agnostic between static and dynamic array sources, using `var` will not allow changing the length of the array passes. If you need to do that, you should type the parameter to either `TArray<T>` or your own custom type instead of an open array:

```
procedure InitializeArray(var Arr: TArray<string>);
begin
  SetLength(Arr, 2);
  Arr[0] := 'No static arrays allowed';
  Arr[1] := 'Thank you very much!';
end;
```

Sorting dynamic arrays

The `System.Generics.Collections` unit provides procedures to both sort and search single dimensional dynamic arrays. These are implemented as generic methods on the `TArray` type. While this is a different type to the `TArray` declared by the `System` unit, it does not conflict — technically, the `System` version is a generic type where `System.Generics.Collections` provides a non-generic class with generic class methods.

The sorting method, `Sort`, is overloaded as thus:

```
class procedure Sort<T>(var Values: array of T);
class procedure Sort<T>(var Values: array of T;
  const Comparer: IComparer<T>);
class procedure Sort<T>(var Values: array of T;
  const Comparer: IComparer<T>; Index, Count: Integer);
```

If the single parameter version is called, the default 'comparer' for `T` is used, and the whole array sorted. The second and third overloads allow using a custom comparer and sorting only part of the array respectively. In all cases, the 'quicksort' algorithm is used behind the scenes to do the actual sorting:

```
uses
  System.Generics.Collections;
```

```
var
  Arr: TArray<string>;
  S: string;
begin
  Arr := TArray<string>.Create('Bill', 'Jane', 'Adam');
  TArray.Sort<string>(Arr);
  for S in Arr do
    Write(S + ' '); //output: Adam Bill Jane
```

For simple types, the default comparer will usually be what you want. A possible exception is the default string comparer, since it is case sensitive. Nonetheless, a case insensitive version is provided by the `System.Generics.Defaults` unit. To use it, pass `TIStringComparer.Ordinal` as the second argument to `Sort`:

```
uses
  System.Generics.Collections, System.Generics.Defaults;

var
  Arr: TArray<string>;
  S: string;
begin
  Arr := TArray<string>.Create('things', 'SOME', 'some', 'THINGS');
  TArray.Sort<string>(Arr);
  for S in Arr do
    Write(S, ' '); //output: SOME THINGS some things
  WriteLn;
  TArray.Sort<string>(Arr, TIStringComparer.Ordinal);
  for S in Arr do
    Write(S, ' '); //output: some SOME things THINGS
```

For structured types like records or classes, the default comparer is likely to be next to useless. In the case of records it does a binary comparison of the raw bytes, ignoring the record's division into fields; with classes, it simply compares references (i.e., memory addresses — an implementation detail in other words).

To define a custom comparer, you need to provide an object implementing the `IComparer<T>` interface, as declared in `System.Generics.Defaults`:

```
IComparer<T> = interface
  function Compare(const Left, Right: T): Integer;
end;
```

The easiest way to do this is to call `TComparer<T>.Construct`, passing an anonymous method with a signature that matches `TComparer<T>.Compare`. However you implement the interface, the comparison function should return the following:

- 0 when the items are equal
- Less than 0 if the first item should come before the second
- More than 0 if the second item should come before the first.

This matches the semantics of string comparison functions such as `CompareStr`.

So, imagine having an array of records, each record holding a person's name and age:

```
uses
  System.Generics.Collections, System.Generics.Defaults;

type
  TPersonRec = record
    Forename, Surname: string;
    Age: Byte;
    constructor Create(const AForename, ASurname: string;
      AAge: Byte);
  end;

constructor TPersonRec.Create(
  const AForename, ASurname: string; AAge: Byte);
begin
  Forename := AForename;
  Surname := ASurname;
  Age := AAge;
end;

var
  Rec: TPersonRec;
  People: TArray<TPersonRec>;
begin
```

```
  SetLength(People, 5);
  People[0] := TPersonRec.Create('Jill', 'Smith', 32);
  People[1] := TPersonRec.Create('Robert', 'Jones', 28);
  People[2] := TPersonRec.Create('Aziz', 'Howayek', 28);
  People[3] := TPersonRec.Create('Anjum', 'Smith', 32);
  People[4] := TPersonRec.Create('Raza', 'Saad', 27);
```

Say you want to sort `People` by age, then surname, then forename. Using the anonymous method approach, this can be done as thus:

```
TArray.Sort<TPersonRec>(People, TComparer<TPersonRec>.Construct(
  function (const L, R: TPersonRec): Integer
  begin
    Result := L.Age - R.Age;
    if Result <> 0 then Exit;
    Result := CompareStr(L.Surname, R.Surname, loUserLocale);
    if Result = 0 then Result := CompareStr(L.Forename,
      R.Forename, loUserLocale);
  end));
```

If you then write out the result like this:

```
for Rec in People do
  WriteLn('Age ', Rec.Age, ': ', Rec.Surname, ', ', Rec.Forename);
```

You get the following output:

```
Age 27: Saad, Raza
Age 28: Howayek, Aziz
Age 28: Jones, Robert
Age 32: Smith, Anjum
Age 32: Smith, Jill
Age 41: Jacobs, Anne
```

Binary and other searches

`TArray.BinarySearch` is overloaded in a similar fashion to `TArray.Sort`:

```
class function BinarySearch<T>(const Values: array of T;
  const Item: T; out FoundIndex: Integer): Boolean;
class function BinarySearch<T>(const Values: array of T;
  const Item: T; out FoundIndex: Integer;
  const Comparer: IComparer<T>): Boolean;
class function BinarySearch<T>(const Values: array of T;
  const Item: T; out FoundIndex: Integer;
  const Comparer: IComparer<T>; Index, Count: Integer): Boolean;
```

A 'binary' search works by repeatedly testing middle values; if the value is too low, then the value in between the one tested and the one at the top of the range is tested next, otherwise the value in between it and the bottom one is, and so on.

Because of the way a binary search works, you need to have sorted the array first:

```
var
  Arr: TArray<string>;
  Index: Integer;
  Matched: Boolean;
begin
  Arr := TArray<string>.Create('beta', 'alpha', 'gamma');
  Matched := TArray.BinarySearch<string>(Arr, 'beta', Index);
  WriteLn(Matched);         //output: FALSE
  TArray.Sort<string>(Arr);
  Matched := TArray.BinarySearch<string>(Arr, 'beta', Index);
  WriteLn(Matched);         //output: TRUE
```

While binary searches are typically fast, you may of course not want to sort an array before searching. In that case, simply walking up the array and testing each element in turn will have to do.

While `TArray` itself does not provide a method to do this, one is easy enough to add via a class helper. Since the default comparer for a type is exposed via `TComparer<T>.Default`, you can make use of the same comparer objects `BinarySearch` and `Sort` use:

```
unit SimpleArraySearch;

interface

uses
  System.SysUtils, System.Generics.Collections,
```

```
  System.Generics.Defaults;

type
  TArrayHelper = class helper for TArray
    class function FindFirst<T>(const Arr: array of T;
      const Item: T; out FoundIdx: Integer): Boolean; overload;
    class function FindFirst<T>(const Arr: array of T;
      const Item: T; out FoundIdx: Integer;
      const Comparer: IComparer<T>): Boolean; overload;
    class function FindFirst<T>(const Arr: array of T;
      const Item: T; out FoundIdx: Integer; const Comparer:
      IComparer<T>; Index, Count: Integer): Boolean; overload;
  end;

implementation

uses System.RTLConsts;

class function TArrayHelper.FindFirst<T>(const Arr: array of T;
  const Item: T; out FoundIdx: Integer): Boolean;
begin
  Result := FindFirst<T>(Arr, Item, FoundIdx,
    TComparer<T>.Default, 0, Length(Arr))
end;

class function TArrayHelper.FindFirst<T>(const Arr: array of T;
  const Item: T; out FoundIdx: Integer;
  const Comparer: IComparer<T>): Boolean;
begin
  Result := FindFirst<T>(Arr, Item, FoundIdx,
    Comparer, 0, Length(Arr))
end;

class function TArrayHelper.FindFirst<T>(const Arr: array of T;
  const Item: T; out FoundIdx: Integer;
  const Comparer: IComparer<T>; Index, Count: Integer): Boolean;
var
  I: Integer;
begin
  //test the bounds, raising an exception if bad
  if (Index < Low(Arr)) or (Index + Count > Length(Arr)) then
    raise EArgumentOutOfRangeException.CreateRes(
      @SArgumentOutOfRange);
  //loop through looking for the item
  for I := Index to Count - 1 do
    if Comparer.Compare(Arr[I], Item) = 0 then
    begin
      FoundIdx := I;
      Result := True;
      Exit;
    end;
  //if still here, couldn't have found it
  Result := False;
end;

end.
```

Other variants like a `FindLast` method might be coded similarly.

Standard collection classes

'Collection classes' are 'list' or 'group of' classes, a bit like dynamic arrays but being much more flexible about adding and removing individual items. In Delphi, the term 'collection' can also be used much more narrowly to refer to `TCollection` specifically, a specialist list-type class that integrates with the VCL's streaming system. This chapter's concern will be with collection classes in the broader sense; since `TCollection` has little purpose it you aren't developing custom components, it will be covered in chapter 8 (the streaming chapter) instead.

TBits

Declared in the `System.Classes` unit, the `TBits` class defines a memory-efficient array-type object for `Boolean` values. Unlike an actual `array of Boolean`, which uses one byte — eight bits — per value, a `TBits` instance will pack up to eight values into every byte of memory used.

The public interface of `TBits` is very simple:

```
function OpenBit: Integer;
property Bits[Index: Integer]: Boolean read GetBit
  write SetBit; default;
property Size: Integer read FSize write SetSize;
```

On creation, `Size` is 0. When set to a value larger than the current one, the new bits are initialised to `False` ('open'). The `Bits` default property then reads or writes a specific value, and `OpenBit` finds the index of the first `False` item.

Here's an example of `TBits` at work, being used to help calculate prime numbers using the 'sieve of Eratosthenes' algorithm:

```
uses System.SysUtils, System.Classes;

const
  MaxNum = 100000;
var
  DoneFirst: Boolean;
  I, J: Integer;
  Numbers: TBits;
begin
  {$OVERFLOWCHECKS OFF}         //Explicitly check for overflows
  Numbers := TBits.Create;
  try
    Numbers.Size := MaxNum + 1; //TBits indexes from zero
    for I := 2 to MaxNum do
      if not Numbers[I] then    //Bits is the default property
      begin
        J := I * I;            //'Start at square' optimisation
        while (J <= MaxNum) and (J > 0) do
        begin
          Numbers[J] := True;
          Inc(J, I);
        end;
      end;
```

Then, to output the results:

```
    Write('From 1 to ', MaxNum,
      ', the prime numbers are the following: ');
    DoneFirst := False;
    for I := 2 to MaxNum do
      if not Numbers[I] then
      begin
        if DoneFirst then
          Write(', ', I)
        else
        begin
          DoneFirst := True;
          Write(I);
        end;
      end;
  finally
    Numbers.Free;
  end;
end.
```

TList

`TList` is a class for maintaining an ordered list of things with a specific type. The Delphi RTL actually defines two `TList` classes, one non-generic (in `System.Classes`) and the other generic (`System.Generics.Collections`). The non-generic version, which defines a list of pointers, is the older class, and should be considered for backwards compatibility only — if generics had been part of the Delphi language from the off, it would never have existed.

When declaring an instance of the generic `TList`, you specify the item type, which can be anything — in other words, its type parameter is unconstrained:

```
var
  List: TList<Integer>; //declare a list of integers
```

The list is then instantiated in the normal way:

```
List := TList<Integer>.Create;
```

Once created, you may choose to immediately set the list's `Capacity` property. This is a particularly good idea in the case of a `TList` of records, or when a lot of items will be added:

```
List.Capacity := 128;
```

While you don't have to set `Capacity`, doing so will avoid the need for memory reallocations as the list grows in size. You don't have to know exactly how many items you will be adding up front — just make a reasonable guess, and the capacity will be increased automatically if the guess proves too low.

To actually add items, call `Add` or (for a group of items) `AddRange` to append to the end of the list. Alternatively, call `Insert` or `InsertRange` to add one or more items at a specific position. Since `TList` is zero indexed, the first item has an index of 0:

```
List.Add(99);                //add an individual item
List.AddRange([45, 83, 13]); //add an array of items
List.Insert(0, 361);         //insert 361 at the top of list
List.InsertRange(1, List2);  //copy from another TList<Integer>
```

To move an item, use either `Move` or `Exchange`. As its name implies, `Exchange` swaps the position of two items, whereas `Move` just moves one item. A `Reverse` method is also available to reverse the order of a list's items in one call:

```
List.Move(0, List.Count - 1); //move the first item to the end
List.Exchange(2, 3);          //switch position of two items
List.Reverse;                 //reverse the order of all items
```

Use `Delete` to remove the item at a specified position, `Remove` to remove an item by specifying the item itself (only the first occurrence will get removed if the item appears more than once), and `Clear` to remove everything:

```
List.Delete(1);             //delete the second item
List.Remove(99);            //remove 99 from the list
List.Clear;                 //remove everything
```

The fact a `TList` object is ordered means you can access items by their index, as if the list were an array:

```
for I := 0 to List.Count - 1 do
  WriteLn(' ', List[I]);
```

Again like an array, you can also enumerate a list using a `for`/`in` loop:

```
for I in List do
  WriteLn(' ', I);
```

`TList` also has `Sort` and `BinarySearch` methods that work in exactly the same way as `TArray.Sort` and `TArray.BinarySearch`. If you wish to use a custom comparer with `Sort` and `BinarySearch`, you can either pass one to the `TList` constructor, or alternatively, pass it as a parameter to `Sort` and `BinarySearch` themselves:

```
{ Sort and search for an item using the default comparer }
List.Sort;
if List.BinarySearch(99, I) then
  WriteLn('99 is now at position ', I)
else
  WriteLn('Couldn''t find 99 in the list');
{ Sort in descending order }
List.Sort(TComparer<Integer>.Construct(
  function (const L, R: Integer): Integer
  begin
    Result := R - L;
  end));
```

As with arrays, using a binary search will be quick, but requires the list to have been sorted first. If you don't want to do that, you can call either `IndexOf` or `LastIndexOf` instead. Neither is overloaded to take a custom comparer though — if required, you will need to have set one via the list's constructor:

```
List := TList<Integer>.Create(TComparer<Integer>.Construct(
  function (const L, R: Integer): Integer
  begin
    Result := R - L;
  end));
```

TObjectList

A variant of `TList` is `TObjectList`. Inheriting from `TList`, it has all the parent class' functionality and behaviour but with one addition and one corresponding limitation: the ability to 'own' the items added to it, and the requirement for the item type to be a class. 'Owning' its items means a `TObjectList` will call an item's `Free` method on it being deleted from the list, whether directly or because the list itself is being cleared or destroyed:

```
type
  TItem = class
  strict private
    FName: string;
  public
    constructor Create(const AName: string);
    destructor Destroy; override;
  end;

constructor TItem.Create(const AName: string);
begin
  inherited Create;
  FName := AName;
end;

destructor TItem.Destroy;
begin
  WriteLn(FName, ' is being destroyed...');
  inherited Destroy;
end;

var
  List: TObjectList<TItem>;
begin
  List := TObjectList<TItem>.Create;
  try
    List.AddRange([TItem.Create('1st'), TItem.Create('2nd')]);
    { Item is freed as it is removed }
    List.Delete(1);
    WriteLn('List.Count = ', List.Count);
  finally
    { Freeing the list will also free any remaining items in it }
    List.Free;
  end;
  WriteLn('Finished');
end.
```

If you want to remove an item *without* freeing it, you should call the list's `Extract` method:

```
List.Extract(SomeObj);
```

When an object is owned by a `TObjectList`, you must be careful not to have it destroyed when you didn't intend it to be. For example, to move an item, you *must* call `Move` or `Exchange` rather than first `Delete` then `Insert` or `Add`. Furthermore, if moving items between two object lists, you must temporally set the first list's `OwnsObjects` property to `False` before clearing it:

```
type
  TMyObject = class
  end;

var
  Source, Dest: TObjectList<TMyObject>;
begin
  Source := TObjectList<TMyObject>.Create;
  Dest := TObjectList<TMyObject>.Create;
  try
    Source.AddRange([TMyObject.Create, TMyObject.Create]);
    Dest.AddRange(Source);
    { As Dest now owns the items, Source shouldn't too }
    Source.OwnsObjects := False;
    { If Source is not to be used any more, just free it,
      otherwise call Clear before resetting OwnsObjects and
      adding the new items to it }
```

```
    Source.Clear;
    Source.OwnsObjects := True;
    Source.Add(TMyObject.Create);
    //...
  finally
    Source.Free;
    Dest.Free;
  end;
end.
```

TComponentList

`TComponentList` is a subclass of `TObjectList` that has knowledge of the `TComponent` ownership pattern. This means that if you add a component to one that then gets freed before the list itself (e.g., you add a reference to button whose form gets freed when it closes, taking the button with it), the list will remove the now stale component reference from itself automatically. In contrast, a regular `TObjectList` would not, potentially causing access violations later on.

Unfortunately, no standard generic version of `TComponentList` exists, so there's just the old non-generic one in `System.Contnrs` to use:

```
uses
  System.Classes, System.Contnrs, Vcl.Forms, Vcl.Dialogs,
  System.Generics.Collections;

var
  CompList: TComponentList;
  S: string;
  StdList: TList<TComponent>;
  TestForm: TForm;
begin
  CompList := TComponentList.Create(False); //Don't own items
  StdList := TList<TComponent>.Create;
  try
    { Create the test form and add it to the two lists. }
    TestForm := TForm.Create(nil);
    CompList.Add(TestForm);
    StdList.Add(TestForm);
    { Confirm it's been added by reporting the Count props. }
    S := Format('CompList.Count = %d' + sLineBreak +
      'StdList.Count = %d', [CompList.Count, StdList.Count]);
    ShowMessage(S);
    { Free the form and see what the Count props now report:
      CompList's will be 0 (good!) and StdList's 1 (whoops!). }
    TestForm.Free;
    S := Format('CompList.Count = %d' + sLineBreak +
      'StdList.Count = %d', [CompList.Count, StdList.Count]);
    ShowMessage(S);
  finally
    CompList.Free;
    StdList.Free;
  end;
end.
```

A generic version of `TComponentList` isn't hard to write. To do so, create a new unit, and add the following to its `interface` section:

```
uses
  System.SysUtils, System.Classes, System.Generics.Collections;

type
  TComponentListProxy = class(TComponent)
  strict private type
    TDeleteNotifyEvent = procedure (Comp: TComponent) of object;
  strict private
    FOnDelete: TDeleteNotifyEvent;
  protected
    procedure Notification(AComponent: TComponent;
      AOperation: TOperation); override;
  public
    constructor Create(const OnDelete:
      TDeleteNotifyEvent); reintroduce;
  end;

  TComponentList<T: TComponent> = class(TList<T>)
  strict private
    FOwnsObjects: Boolean;
```

```
    FProxy: TComponentListProxy;
    procedure ComponentDeleted(Component: TComponent);
  protected
    procedure Notify(const Item: T;
      Action: TCollectionNotification); override;
  public
    destructor Destroy; override;
    property OwnsObjects: Boolean read FOwnsObjects
      write FOwnsObjects;
  end;
```

The idea here — used by the stock `TComponentList` too — is to set up a proxy component that can receive notification of when an item has been freed. The list can't be notified directly, since only fellow `TComponent` descendants can partake in the `TComponent` free notification system.

Here's how the proxy and the main class are implemented:

```
constructor TComponentListProxy.Create(
  const OnDelete: TDeleteNotifyEvent);
begin
  inherited Create(nil);
  FOnDelete := OnDelete;
end;

procedure TComponentListProxy.Notification(AComponent:
  TComponent; AOperation: TOperation);
begin
  if AOperation = opRemove then FOnDelete(AComponent);
end;

destructor TComponentList<T>.Destroy;
begin
  FreeAndNil(FProxy);
  inherited;
end;

procedure TComponentList<T>.ComponentDeleted(Component: TComponent);
begin
  Remove(Component);
end;

procedure TComponentList<T>.Notify(const Item: T;
  Action: TCollectionNotification);
begin
  inherited;
  case Action of
    cnAdded:
      begin
        if FProxy = nil then
          FProxy := TComponentListProxy.Create(ComponentDeleted);
        TComponent(Item).FreeNotification(FProxy);
      end;
    cnRemoved, cnExtracted:
    begin
      if FProxy <> nil then Item.RemoveFreeNotification(FProxy);
      if OwnsObjects and (Action = cnRemoved) then Item.Free;
    end;
  end;
end;
```

Once written, the generic `TComponentList` is used in exactly the same way as the non-generic version, just with the ability to enforce specialising to something more specific than `TComponent`.

TStack and TObjectStack

A 'stack' is a list in which items are 'pushed' onto the end then 'popped' back off in 'LIFO' order (last in, first out). Think of how a grit bin is used during snowy weather: the grit that lies at the top of the bin will always be taken out first, entailing the grit that went in first coming out last, or never at all if the bin is always promptly topped up.

Similar to its provision of both `TList` and `TObjectList`, `System.Generics.Collections` provides two stack classes, `TStack` and `TObjectStack`, the latter owning its items in the same way `TObjectList` does.

Here's an example of `TStack` in action. In it, `TStack` is used to implement an `IntToStr` variant that outputs Eastern Arabic numerals rather than the Western Arabic set used by modern European languages:

```
function IntToEasternArabicStr(const Value: Int64): string;
const
  NumberChars: array[0..9] of Char = ('٣' ,'٢' ,'١' ,'٠',
    '٩' ,'٨' ,'٧' ,'٦' ,'٥' ,'٤');
var
  CurValue: Int64;
  I: Integer;
  Stack: TStack<Char>;
begin
  if Value = 0 then Exit(NumberChars[0]);
  Stack := TStack<Char>.Create;
  try
    if Value > 0 then
      CurValue := Value
    else
      CurValue := -Value;
    repeat
      Stack.Push(NumberChars[CurValue mod 10]);
      CurValue := CurValue div 10;
    until (CurValue = 0);
    if Value < 0 then Stack.Push('-');
    SetLength(Result, Stack.Count);
    for I := 1 to Stack.Count do
      Result[I] := Stack.Pop;
  finally
    Stack.Free;
  end;
end;
```

In this particular example, every item that will get pushed onto the stack is pushed at the start. You don't have to use a stack like that though— as with a grit bin, you can alternate between pushing (refilling) and popping (taking out) instead.

If you have a lot of items to push, what was said about setting the `Capacity` property of a `TList` object applies equally (perhaps more so) to `TStack` too. This is because the stack class' default behaviour is to have an initial capacity of only 4 items, being doubled each time more space is needed.

TQueue and TObjectQueue

A 'queue' is very similar to a stack, only with its items 'popped' or 'dequeued' in the order they were 'pushed' or 'enqueued' (FIFO — first in, first out). Similar to before, `System.Generics.Collections` provides both `TQueue` and `TObjectQueue` versions.

In essence, using a queue is like using a regular list when you only add items to the end of it and only remove items from the top of it. As a result, `TestList` and `TestQueue` in the following code do exactly the same thing:

```
const
  Iterations = 1000;

type
  TDatum = array[1..10024] of Byte; //10K of bytes

var
  DummyDatum: TDatum;

procedure TestList;
var
  I: Integer;
  List: TList<TDatum>;
begin
  List := TList<TDatum>.Create;
  try
    List.Capacity := Iterations;
    for I := 1 to Iterations do
      List.Add(DummyDatum);
    for I := 1 to Iterations do
    begin
      DummyDatum := List[0];
      List.Delete(0);
    end;
  finally
    List.Free;
  end;
end;

procedure TestQueue;
```

```
var
  I: Integer;
  Queue: TQueue<TDatum>;
begin
  Queue := TQueue<TDatum>.Create;
  try
    Queue.Capacity := Iterations;
    for I := 1 to Iterations do
      Queue.Enqueue(DummyDatum);
    for I := 1 to Iterations do
      DummyDatum := Queue.Dequeue;
  finally
    Queue.Free;
  end;
end;
```

Nevertheless, there is one significant difference here: the `TQueue` version is much, much faster. The reason is that every time you remove an item from the head of a `TList`, every other item has to be moved down a place in memory. In contrast, `TQueue` does some internal bookmarking that avoids having to do this.

Naturally, the test code just presented is written to make the difference very pronounced — the `TList` is being made to shuffle around a lot of data for every dequeue — but it neatly demonstrates the potential difference in performance. On my computer, for example, the `TQueue` version is well over *100 times* quicker!

TDictionary and TObjectDictionary

A 'dictionary' in a programming sense (sometimes called an 'associative array') maps 'keys' to 'values'. In any given dictionary, no one key can appear twice. Usually the values are unique too, however in principle, you can have the same value mapped to different keys, similar to how different words can have the same meaning. Nevertheless, the analogy with a dictionary in the ordinary sense of the term quickly falls down, since each key in a Delphi dictionary has exactly one value, in contrast to the varying number of meanings a word can have. Furthermore, neither keys nor values need be strings (textual items) — instead, any type can be used, so long as all the keys have one type and all the values another, be this the same type as the key one or not.

Following the pattern of the list classes, `System.Generics.Collections` provides two dictionary classes, `TDictionary` and `TObjectDictionary`. In use, you specify the key and value types on declaring a dictionary variable. For example, the following declares a string/procedural pointer dictionary. This will allow looking up a routine via some sort of textual name or description:

```
type
  TProcedure = procedure;

var
  ProcMap: TDictionary<string, TProcedure>;
```

While you don't have to define one, sometimes it can be more readable to use a type alias:

```
type
  TStringControlMap = TObjectDictionary<string, TControl>;

var
  ObjDict1, ObjDict2, ObjDict3: TStringControlMap;
```

When creating a dictionary, you can specify the starting capacity, a custom 'equality comparer' (more on which shortly), and/or another dictionary to copy items from. In the case of a `TObjectDictionary`, you can also specify whether the dictionary owns the keys, the values, or both. Be careful not to request the dictionary own things that aren't class instances however:

```
ObjDict1 := TStringControlMap.Create([doOwnsValues]); //OK
ObjDict2 := TStringControlMap.Create([doOwnsKeys]); //bad!
ObjDict3 := TStringControlMap.Create([doOwnsKeys, doOwnsValues]); //bad!
```

Alternately, both `TDictionary` and `TObjectDictionary` can be constructed with no arguments passed:

```
ProcMap := TDictionary<string, TProcedure>.Create;
```

Then, in use:

```
procedure Foo;
begin
  WriteLn('Foo says hi');
end;

procedure Bar;
```

```
begin
  WriteLn('Bar says hi');
end;

var
  Proc: TProcedure;
  S: string;
begin
  ProcMap.Add('Foo', Foo);
  ProcMap.Add('Bar', Bar);
  Write('Enter the name of procedure to call: ');
  ReadLn(S);
  if S = '' then Exit;
  if ProcMap.TryGetValue(S, Proc) then
    Proc
  else
    WriteLn('"', S, '" is not a registered procedure!');
```

This code adds a couple of items to the dictionary, before asking the user to specify the name of a procedure to call. Once entered, the dictionary is used to map the name to an actual procedure, and the procedure invoked.

Using dictionaries

To add a new item to the dictionary, call either `Add`, like in the previous example, or `AddOrSetValue`. The difference between the two is what happens when the specified key has already been set: where `Add` will raise an exception, `AddOrSetValue` will just replace the existing value.

Because of this, `AddOrSetValue` can also be used to change the value of an existing key. If the key should already exist though, you can also index the dictionary as if it were an array:

```
ProcMap['Foo'] := NewFoo;
```

The same array-like syntax works to retrieve an item too:

```
ProcToCall := ProcMap['Fum'];
```

If the key does not exist however, using the array syntax will raise an exception. To only speculatively retrieve a value, you should therefore use the `TryGetValue` function instead:

```
if ProcMap.TryGetValue('Fum', Value) then
  WriteLn('Got it')
else
  Writeln('Could not find it');
```

To test whether a key exists in the first place, call `ContainsKey`. A `ContainsValue` function also exists to test whether a certain value appears at least one time in the dictionary:

```
if ProcMap.ContainsKey('Bar') then
  WriteLn('"Bar" exists as a key');
if ProcMap.ContainsValue(Fum) then
  WriteLn('The "Fum" procedure has been added as a value');
```

Equality comparers

As with lists, stacks and queues, dictionaries can be given a custom comparer to use. However, this is not for the purposes of sorting, but for how the dictionary behaves on simple additions, specifying what is to count as the ‘same’ key. A dictionary comparer also has its own special interface, `IEqualityComparer<T>`:

```
type
  IEqualityComparer<T> = interface
    function Equals(const Left, Right: T): Boolean;
    function GetHashCode(const Key: T): Integer;
  end;
```

`Equals` should report whether the two values are the ‘same’, and `GetHashCode` must provide an integral code that, as much as possible, uniquely identifies the given key. You can if you wish use the default hashing algorithm, which is exposed by `System.Generics.Defaults` as the `BobJenkinsHash` function. This is used for almost all types internally; the main exception is when a class is used for the key type, in which case the key objects’ `GetHashCode` function is used. This will simply cast `Self` to `Integer` by default.

Like how a custom `IComparer` implementation can be written by passing an anonymous method to `TComparer<T>.Construct`, so a custom `IEqualityComparer` implementation can be constructed by passing a pair of anonymous methods to `TEqualityComparer<T>.Construct`, the first for `Equals` and the second for `GetHashCode`.

However you implement the interface, the return values of `Equals` and `GetHashCode` must correspond to each other. Thus, if writing a custom comparer for a record type that looks at actual field values, it is not enough to provide a custom `Equals` implementation but delegate to `BobJenkinsHash` in a way that hashes the record as a whole — you need to implement `GetHashCode` properly too.

The easiest case will be if hashing against a single field value. For example, the following sets up a class to be a dictionary key type by comparing and hashing against an `Int64` field:

```
uses System.SysUtils, System.Generics.Defaults;

type
  TInt64Object = class
    Data: Int64;
    function Equals(Obj: TObject): Boolean; override;
    function GetHashCode: Integer; override;
  end;

function TInt64Object.Equals(Obj: TObject): Boolean;
begin
  if Obj = Self then
    Result := True
  else if not (Obj is TInt64Object) then
    Result := False
  else
    Result := TInt64Object(Obj).Data = Data;
end;

function TInt64Object.GetHashCode: Integer;
begin
  Result := BobJenkinsHash(Data, SizeOf(Data), 0);
end;
```

Even working against a single field can have its subtleties however. In particular, aiming to do this with a string field can easily give rise to inconsistencies if `Equals` deals with Unicode quirks like decomposed characters while `GetHashCode` does not. This will be the case if you delegate to either `CompareStr(S1, S2, loUserLocale)` or `CompareText(S1, S2, loUserLocale)` for `Equals` and `BobJenkinsHash` for `GetHashCode`, given `BobJenkinsHash` sees its input as a sequence of raw bytes where `CompareText` understands it to be a sequence of Unicode characters.

One way round this particular problem is have `Equals` call not `CompareText`, or even `CompareStr` after converting both inputs to lower case, but `CompareMem` after doing so. For example, imagine the following data structures:

```
type
  TNameRec = record
    Forename, Surname: string;
    constructor Create(const AForename, ASurname: string);
  end;

constructor TNameRec.Create(const AForename, ASurname: string);
begin
  Forename := AForename;
  Surname := ASurname;
end;

var
  NameToAgeMap: TDictionary<TNameRec,Integer>;
```

Say we want `NameToAgeMap` to compare keys by looking at the `Forename` and `Surname` fields case insensitively. Similar to what is available for arrays and lists, a stock case insensitive dictionary comparer is provided by the `TIStringComparer.Ordinal` function. However, it cannot be used in our case because the key is a record, not a string. So, we need to implement the custom comparer ourselves.

Once you need to provide two methods, as we do now, it is clearer to define an explicit helper class that implements `IEqualityComparer<T>` directly, so let's do that:

```
type
  TMyEqualityComparer = class(TInterfacedObject,
    IEqualityComparer<TNameRec>)
    function Equals(const L, R: TNameRec): Boolean;
    function GetHashCode(const Value: TNameRec): Integer;
  end;

function TMyEqualityComparer.Equals(const L, R: TNameRec): Boolean;
var
  S1, S2: string;
```

```
begin
  S1 := LowerCase(L.Forename + L.Surname, loUserLocale);
  S2 := LowerCase(R.Forename + R.Surname, loUserLocale);
  Result := (Length(S1) = Length(S2)) and
    CompareMem(PChar(S1), PChar(S2), ByteLength(S1));
end;

function TMyEqualityComparer.GetHashCode(const Value: TNameRec): Integer;
var
  S: string;
begin
  S := LowerCase(Value.Forename + Value.Surname, loUserLocale);
  Result := BobJenkinsHash(PChar(S)^, ByteLength(S), 0);
end;
```

The comparer returned by `TIStringComparer.Ordinal` first converts everything to lower case before treating the results as raw sequences of bytes in both `Equals` and `GetHashCode`; we do the same here. While there may be edge cases where this produces a negative result where `CompareText` would find a match, it ensures internal consistency.

Now written, our custom comparer can be used as thus:

```
var
  NameToAgeMap: TDictionary<TNameRec,Integer>;
  Name: TNameRec;
begin
  NameToAgeMap := TDictionary<TNameRec,Integer>.Create(
    TMyEqualityComparer.Create);
  try
    //add some items
    NameToAgeMap.Add(TNameRec.Create('John', 'Smith'), 48);
    NameToAgeMap.Add(TNameRec.Create('Joe', 'Bloggs'), 59);
    //test whether there's a Joe Smith in there
    Name := TNameRec.Create('JOE', 'SMITH');
    WriteLn(NameToAgeMap.ContainsKey(Name)); //output: FALSE
    //test whether there's a Joe Bloggs in there
    Name.Surname := 'BLOGGS';
    WriteLn(NameToAgeMap.ContainsKey(Name)); //output: TRUE
  finally
    NameToAgeMap.Free;
  end;
```

Enumerating a TDictionary

`TDictionary` supports enumerating its items in three ways: with respect to its keys via the `Keys` property, its values via the `Values` property, and its keys-and-values in the form of `TPair` records. The last of these is done via a `for`/`in` loop directly on the dictionary object itself —

```
var
  Key: string;
  Value: TProcedure;
  Item: TPair<string, TProcedure>; //match dictionary's type args
begin
  //NB: this continues the string-procedural pointer example...
  WriteLn('Keys:');
  for Key in ProcMap.Keys do
    WriteLn('  ', Key);

  WriteLn('Values expressed as procedural addresses:');
  for Value in ProcMap.Values do
    WriteLn(Format('  $%p', [@Proc]));

  WriteLn('Key/value pairs:');
  for Item in ProcMap do
    WriteLn(Format('  %s = $%p', [Item.Key, @Item.Value]));
```

Since `TDictionary` is not an 'ordered' collection, you should consider the sequence in which items are enumerated to be essentially random. Thus, consider the following program:

```
uses
  System.Generics.Collections;

var
  Dict: TDictionary<string, string>;
  Pair: TPair<string, string>;
begin
  Dict := TDictionary<string, string>.Create;
```

```
  try
    Dict.Add('4th', 'Arsenal');
    Dict.Add('3rd', 'Manchester City');
    Dict.Add('2nd', 'Chelsea');
    Dict.Add('1st', 'Manchester United');
    for Pair in Dict do
      WriteLn(Pair.Key, ': ', Pair.Value);
  finally
    Dict.Free;
  end;
end.
```

When cycling though the contents, you are likely to get items neither in the order they were added, nor in case order. I get this output:

```
3rd: Manchester City
4th: Arsenal
2nd: Chelsea
1st: Manchester United
```

Sometimes this can be unacceptable, and unfortunately, the standard RTL does not provide an ordered dictionary class as an alternative to `TDictionary`. If you don't mind using third party code, such a class is nevertheless available in the 'Collections' open source project of Alexandru Ciobanu, a former member of the Delphi RTL team (`http://code.google.com/p/delphi-coll/`). Alternatively, the `TStrings` class — and in particular, the `THashedStringList` class of `System.IniFiles` — can serve as an 'in the box' solution for string-string or (at a pinch) string-object mappings.

THashedStringList

As its name suggests, `THashedStringList` is a `TStringList` descendant, and so implements the standard `TStrings` interface discussed in the previous chapter. Unlike a normal `TStringList` though, it uses a hashing system to speed up searches.

On first glance, the fact `THashedStringList` is implemented by the `System.IniFiles` unit may suggest it is specific to INI files, but that is not the case — it works perfectly well on its own. Its main limitation is that its internal hash is invalidated every time a new string is added. This means that if you repeatedly change the list's contents in between testing for items, it will actually be slower than a regular `TStringList`. However, if items are added up front and rarely changed after that, it works well.

To use `THashedStringList` as a string-string dictionary, add and retrieve items as name=value pairs using the normal `TStrings` interface for that. Unlike in the case of `TDictionary`, indexing an item that doesn't already exist is OK, and in fact the way how you should set an item in the first place (i.e., don't call `Add`):

```
var
  Dict: THashedStringList;
  I: Integer;
begin
  Dict := THashedStringList.Create;
  try
    //load items
    Dict.Values['4th'] := 'Arsenal';
    Dict.Values['3rd'] := 'Manchester City';
    Dict.Values['2nd'] := 'Chelsea';
    Dict.Values['1st'] := 'Manchester United';
    //this will output in the order items were added
    for I := 0 to Dict.Count - 1 do
      WriteLn(Dict.Names[I], ': ', Dict.ValueFromIndex[I]);
    //change an item
    Dict.Values['4th'] := 'Tottenham Hotspur';
    //read it
    WriteLn('Fourth is now ' + Dict.Values['4th']);
```

If you wish to sort a `THashedStringList` in name=value mode, a custom sort function will be required because the standard one compares whole items, not just the 'name' part:

```
function CompareKeys(List: TStringList; Index1,
  Index2: Integer): Integer;
begin
  Result := CompareText(List.Names[Index1],
    List.Names[Index2], loUserLocale);
end;

var
  Dict: THashedStringList;
  I: Integer;
begin
```

```
  Dict := THashedStringList.Create;
  try
    //add items as before, but now sort them...
    Dict.CustomSort(CompareKeys);
    for I := 0 to Dict.Count - 1 do
      WriteLn(Dict.Names[I], ': ', Dict.ValueFromIndex[I]);
```

Using a `THashedStringList` as a string-object dictionary is a little more involved due to the fact `THashedStringList` is not a generic class, and so, does not parameterise its sub-object type. In practical terms, this necessities using typecasting where none is required with a 'real' dictionary. So, say you have defined the following class, which stores details of a racing car team:

```
type
  TTeamDetails = class
    Driver1, Driver2, EngineSupplier: string;
    constructor Create(const ADriver1, ADriver2,
      AEngineSupplier: string);
  end;

constructor TTeamDetails.Create(const ADriver1, ADriver2,
  AEngineSupplier: string);
begin
  inherited Create;
  Driver1 := ADriver1;
  Driver2 := ADriver2;
  EngineSupplier := AEngineSupplier;
end;
```

If you want to create an ordered map of team names to team details, `THashedStringList` might be used for the purpose like this:

```
var
  Details: TTeamDetails;
  I: Integer;
  Map: THashedStringList;
begin
  Map := THashedStringList.Create;
  try
    Map.OwnsObjects := True;
    //load the items
    Map.AddObject('McLaren', TTeamDetails.Create(
      'Lewis Hamilton', 'Jenson Button', 'Mercedes'));
    Map.AddObject('Red Bull', TTeamDetails.Create(
      'Sebastian Vettel', 'Mark Webber', 'Renault'));
    Map.AddObject('Ferrari', TTeamDetails.Create(
      'Fernando Alonso', 'Felipe Massa', 'Ferrari'));
    //enumerate – will come out in order were put in
    for I := 0 to Map.Count - 1 do
    begin
      Details := Map.Objects[I] as TTeamDetails;
      WriteLn(Map[I] + ': ', Details.Driver1, ' and ',
        Details.Driver2);
    end;
    //test whether items exists
    if Map.IndexOf('Virgin') < 0 then
      WriteLn('The Virgin team does not exist');
    //get an item by its 'key'
    Details := Map.Objects[Map.IndexOf('Mclaren')] as TTeamDetails;
    WriteLn('McLaren use a ', Details.EngineSupplier, ' engine');
    //sort the 'keys' and enumerate again
    Map.Sort;
    Write('Now sorted:');
    for I := 0 to Map.Count - 1 do
      Write(' ' + Map[I]);
  finally
    Map.Free;
  end;
end.
```

Going further: linked lists and custom enumerators

Writing linked lists in Delphi

All of the standard list, stack and queue classes are implemented using a dynamic array internally, in which the value of the object's `Capacity` property reflects the length of the underlying array. This implementation makes indexing very fast, and is typically the best way of implementing lists all told.

Nonetheless, it is not the only possible approach, with the main alternative being to write a 'linked list'. This is composed of a series of nodes, each node being made up of a pointer to the next node, some data, and (potentially) a pointer to the previous node too — if so, then the list is a 'doubly-linked list', if not, a 'singly-linked list'.

Unlike an array-based list, a linked list has no concept of a 'capacity' beyond system memory limits. In terms of performance, it can also more efficient to insert into, since there is never any need to 'shuffle along' a whole series of items just to slot in a new one in front of them. On the other hand, indexing a linked list is slow, since to retrieve the *n*th item, you must traverse the list, working your way up *n* nodes. This contrasts with an array-based list, which can index the *n*th item almost directly.

In Delphi, linked lists are things you must implement yourself. However, the basic pattern to follow is simple enough. So, say you want to implement a doubly-linked list of strings. First the node type needs declaring; this can be done with records and pointers, but for simplicity, let's use a class:

```
type
  TNode = class
    Prev, Next: TNode;
    Data: string;
  end;
```

Somewhere there needs to be a variable to store the last node added:

```
var
  LastNode: TNode;
```

To add an item to the list, create a new instance of `TNode`, assign the data, and hook it onto the previous node, fixing up the links as you go:

```
function AddNode(const AData: string): TNode;
begin
  Result := TNode.Create;
  Result.Data := AData;
  Result.Prev := LastNode;
  if LastNode <> nil then LastNode.Next := Result;
  LastNode := Result;
end;
```

Deleting a node requires the same relinking, only done in reverse:

```
procedure DeleteNode(ANode: TNode);
begin
  if ANode.Prev <> nil then ANode.Prev.Next := ANode.Next;
  if ANode.Next <> nil then ANode.Next.Prev := ANode.Prev;
  if LastNode = ANode then LastNode := ANode.Prev;
  ANode.Free;
end;
```

Clearing the list can be done simply by repeatedly deleting the last node until there is no 'last node' to delete:

```
procedure ClearNodes;
begin
  while LastNode <> nil do DeleteNode(LastNode);
end;
```

Then, in use:

```
var
  TopNode, Node: TNode;
begin
  TopNode := AddNode('First');
  AddNode('Second');
  AddNode('Third');
  { Enumerate }
  Node := TopNode;
  repeat
    WriteLn(Node.Data);
    Node := Node.Next;
```

```
  until Node = nil;
  { Delete a node and enumerate again to check the DeleteNode
    logic is correct }
  DeleteNode(TopNode.Next);
  Node := TopNode;
  repeat
    WriteLn(Node.Data);
    Node := Node.Next;
  until Node = nil;
  { Clean up }
  ClearNodes;
end.
```

While 'raw' linked lists like this are easy enough to write, they are also easy to mess up in practice due to the fact the linked list logic is not encapsulated in any way. To fix that, you need some sort of wrapper class. The interface for such a beast might look like this:

```
type
  TLinkedList<T> = class
  public type
    TNode = class
    strict private
      FPrevious, FNext: TNode;
      FList: TSimpleLinkedList<T>;
      FValue: T;
    protected
      constructor Create(AList: TLinkedList<T>;
        APrevious, ANext: TNode; const AValue: T); overload;
    public
      destructor Destroy; override;
      property List: TLinkedList<T> read FList;
      property Next: TNode read FNext write FNext;
      property Previous: TNode read FPrevious write FPrevious;
      property Value: T read FValue write FValue;
    end;
  private
    FFirst, FLast: TNode;
    procedure ValidateNode(ANode: TNode);
  public
    destructor Destroy; override;
    procedure Clear;
    function Add(const AValue: T): TNode; inline;
    function AddAfter(ANode: TNode; const AValue: T): TNode;
    function InsertBefore(ANode: TNode; const AValue: T): TNode;
    property First: TNode read FFirst;
    property Last: TNode read FLast;
  end;
```

Here, the node class has become a nested type, with both inner and outer type encapsulating their state as much as possible. Since you don't want to be repeatedly writing the same logic for different sorts of item type, generics are also used.

In implementing the revised `TNode`, its protected constructor takes the substance of what was in `AddNode` previously; similarly, the destructor takes the substance of what was in `DeleteNode`:

```
constructor TLinkedList<T>.TNode.Create(AList: TLinkedList<T>;
  APrevious, ANext: TNode; const AValue: T);
begin
  inherited Create;
  FPrevious := APrevious;
  FNext := ANext;
  if APrevious <> nil then APrevious.Next := Self;
  if ANext <> nil then ANext.Previous := Self;
  if APrevious = AList.Last then AList.FLast := Self;
  if ANext = AList.First then AList.FFirst := Self;
  FList := AList;
  FValue := AValue;
end;

destructor TLinkedList<T>.TNode.Destroy;
begin
  if Self = List.First then List.FFirst := Next;
  if Self = List.Last then List.FLast := Previous;
  if Previous <> nil then Previous.FNext := Next;
  if Next <> nil then Next.FPrevious := Previous;
  inherited Destroy;
```

```
end;
```

The code for `TLinkedList` itself can be written as thus:

```
destructor TLinkedList<T>.Destroy;
begin
  Clear;
  inherited Destroy;
end;

function TLinkedList<T>.Add(const AValue: T): TNode;
begin
  Result := TNode.Create(Self, Last, nil, AValue);
end;

function TLinkedList<T>.AddAfter(ANode: TNode;
  const AValue: T): TNode;
begin
  ValidateNode(ANode);
  Result := TNode.Create(Self, ANode, ANode.Next, AValue);
end;

procedure TLinkedList<T>.Clear;
begin
  while Count > 0 do First.Free;
end;

function TLinkedList<T>.InsertBefore(ANode: TNode;
  const AValue: T): TNode;
begin
  ValidateNode(ANode);
  Result := TNode.Create(Self, ANode.Previous, ANode, AValue);
end;

procedure TLinkedList<T>.ValidateNode(ANode: TNode);
begin //Should raise a proper exception in practice!
  Assert((Node <> nil) and (ANode.List = Self));
end;
```

The example code for the 'raw' linked list can then be rewritten to use the generic class like this:

```
var
  List: TLinkedList<string>;
  Node: TLinkedList<string>.TNode;
begin
  List := TLinkedList<string>.Create;
  try
    List.Add('First');
    List.Add('Second');
    List.Add('Third');
    { Enumerate }
    Node := List.First;
    repeat
      WriteLn(Node.Value);
      Node := Node.Next;
    until Node = nil;
    { Delete node and enumerate again to check TNode.Destroy logic }
    List.First.Next.Free;
    Node := List.First;
    repeat
      WriteLn(Node.Value);
      Node := Node.Next;
    until Node = nil;
  finally
    List.Free;
  end;
end.
```

In the book's accompanying source code, you can find a 'full fat' version of this class that implements optional object ownership, `TList`-style methods such as `AddRange` and `Remove`, `IComparer`-based searching, and more.

Custom enumerators

We've met the `for/in` loop a few times before, both in this chapter and earlier ones. This has been in the context of merely using an existing enumerator though, whether built into the language (arrays, sets and strings) or provided by the class being enumerated (`TList` etc.).

It is also possible to define custom enumerators that enable `for/in` loops where they are not supported by default. This involves implementing a simple pattern, and is something any class, record or interface can do — in fact, it is only by implementing the enumerable pattern that `TList` and other stock classes actually support `for/in` themselves.

The pattern itself goes as follows: on the type `for/in` support is to be added to, a public parameterless method called `GetEnumerator` must be declared. This must return an object, record or interface that has —

- A public parameterless `Boolean` function called `MoveNext`;
- A public read-only property called `Current` that returns the currently-enumerated element.

When using `for/in` against anything but a set, array or string, your code is therefore shorthand for the following:

```
var
  Enumerator: EnumeratorType;
  Element: ElementType;
begin
  Enumerator := Enumerable.GetEnumerator;
  while Enumerator.MoveNext do
  begin
    Element := Enumerator.Current;
    //body of for/in loop here...
  end;
```

If the enumerator is itself a class, there's an implicit `try/finally` too.

Whether you use a class, record or interface to implement the enumerator is entirely up to you — in particular, there is no requirement for a class to have a class enumerator, a record a record enumerator and so on. Since it will not require the `try/finally` block, a record will be marginally more efficient, however certain RTTI-based frameworks (most notably XE2's LiveBindings) rather lazily assume a custom enumerator will be a class, which may guide your choice.

Custom enumerators by delegation

The simplest way to implement a custom enumerator is to expose an existing one from an encapsulated object. For example, say you have a list-type class, `TWidgetContainer`, that contains numerous `TWidget` objects. For storage, a `TObjectList` is used:

```
uses System.SysUtils, System.Generics.Collections;

type
  TWidget = class
  strict private
    FData: string;
  public
    constructor Create(const AData: string);
    property Data: string read FData;
  end;

  TWidgetContainer = class
  strict private
    FWidgets: TObjectList<TWidget>;
  public
    constructor Create;
    destructor Destroy; override;
  end;

constructor TWidget.Create(const AData: string);
begin
  inherited Create;
  FData := AData;
end;

constructor TWidgetContainer.Create;
begin
  inherited Create;
  FWidgets := TObjectList<TWidget>.Create;
  FWidgets.Add(TWidget.Create(Self, 'First'));
  FWidgets.Add(TWidget.Create(Self, 'Second'));
  FWidgets.Add(TWidget.Create(Self, 'Third'));
end;

destructor TWidgetContainer.Destroy;
begin
  FWidgets.Free;
  inherited Destroy;
```

```
end;
```

Given all the list and dictionary classes in `System.Generics.Collections` have enumerators that inherit from the abstract `TEnumerator<T>` class, our `TWidgetContainer` can implement `GetEnumerator` by simply passing on the request to the `GetEnumerator` method of the encapsulated `TObjectList`:

```
//...
  TWidgetContainer = class
  //...
  public
    function GetEnumerator: TEnumerator<TSubObject>;
  end;

//...

function TWidgetContainer.GetEnumerator: TEnumerator<TWidget>;
begin
  Result := FWidgets.GetEnumerator;
end;
```

With this addition, the following code will compile and output `First`, `Second` and `Third`:

```
var
  Container: TWidgetContainer;
  Widget: TWidget;
begin
  Container := TWidgetContainer.Create;
  try
    for Widget in Container do
      WriteLn(Widget.Data);
  finally
    Container.Free;
  end;
end.
```

Were you to change `TContainedObject` to use a dictionary rather than a list internally, only the implementation of `GetEnumerator` would need changing:

```
  TWidgetContainer = class
  strict private
    FWidgets: TObjectDictionary<string, TWidget>;

//...

function TWidgetContainer.GetEnumerator: TEnumerator<TWidget>;
begin
  Result := FWidgets.Values.GetEnumerator;
end;
```

Implementing a custom enumerator from scratch

To see how implementing an enumerator from scratch works, let's add `for`/`in` support to the `TLinkedList` class presented earlier. The first job is to declare the enumerator type; to keep things clean, this will be nested, just like the `TNode` class already was:

```
type
  TLinkedList<T> = class
  public type
    TNode = class
    //...
    end;

    TEnumerator = record
    strict private
      FDoneFirst: Boolean;
      FNode: TNode;
      function GetCurrent: T;
    public
      constructor Create(AFirst: TNode);
      function MoveNext: Boolean;
      property Current: T read GetCurrent;
    end;
  private
  //...
  public
    function GetEnumerator: TEnumerator;
  //...
```

The enumerator's methods can be implemented as thus:

```
constructor TLinkedList<T>.TEnumerator.Create(AFirst: TNode);
begin
  FDoneFirst := False;
  FNode := AFirst;
end;

function TLinkedList<T>.TEnumerator.GetCurrent: T;
begin
  Result := FNode.Value;
end;

function TLinkedList<T>.TEnumerator.MoveNext: Boolean;
begin
  if FDoneFirst then
    FNode := FNode.Next
  else
    FDoneFirst := True;
  Result := (FNode <> nil);
end;
```

`GetEnumerator` simply returns a new `TEnumerator` instance:

```
function TLinkedList<T>.GetEnumerator: TEnumerator;
begin
  Result := TEnumerator.Create(FFirst);
end;
```

Custom enumerator complete, it can then be used in code like the following:

```
var
  List: TLinkedList<string>;
  S: string;
begin
  List := TLinkedList<string>.Create;
  try
    List.Add('First');
    List.Add('Second');
    List.Add('Third');
    List.Add('Fourth');
    for S in List do
      Write(S + ' ');   //output: First Second Third Fourth
  finally
    List.Free;
  end;
end.
```

Function enumerators

Another variant on the custom enumerator theme is to enable `for`/`in` against not an object, but a standalone function. This might be used in the context of implementing a simple parser of some sort.

For example, a custom enumerator could be defined to read a stream or file in chunks, given a certain file format. For illustrative purposes, let us simply write a `for`/`in` interface for parsing a file or stream in fixed-sized chunks. This will enable writing the following sort of code:

```
var
  Chunk: TBytes;
begin
  //enumerate the file's contents in chunks of 8KB
  for Chunk in ParseChunks('C:\SomeFile.txt', 1024 * 8) do
    WriteLn(StringOf(Chunk));
end.
```

For this to work, `ParseChunks` must return a type with a `GetEnumerator` method. Since the level of indirection (first a type with `GetEnumerator`, then the enumerator itself) is rather artificial in the standalone function case, we will put all methods on the same interface type, with `GetEnumerator` simply returning `Self`:

```
uses
  System.SysUtils, System.Classes;

type
  IChunkParser = interface
    function GetCurrent: TBytes;
    function GetEnumerator: IChunkParser;
    function MoveNext: Boolean;
```

```
    property Current: TBytes read GetCurrent;
  end;

  TChunkSize = 1..High(Integer);

function ParseChunks(AStream: TStream; AChunkSize: TChunkSize;
  AOwnsStream: Boolean = False): IChunkParser; overload;
function ParseChunks(const AFileName: string;
  AChunkSize: TChunkSize): IChunkParser; overload; inline;
```

The interface can be implemented like this:

```
type
  TChunkParser = class(TInterfacedObject, IChunkParser)
  strict protected
    FBuffer: TBytes;
    FOwnsStream: Boolean;
    FStream: TStream;
    function GetCurrent: TBytes;
    function GetEnumerator: IChunkParser;
    function MoveNext: Boolean;
  public
    constructor Create(AStream: TStream; AChunkSize: TChunkSize;
      AOwnsStream: Boolean);
    destructor Destroy; override;
  end;

constructor TChunkParser.Create(AStream: TStream;
  AChunkSize: TChunkSize; AOwnsStream: Boolean);
begin
  inherited Create;
  SetLength(FBuffer, AChunkSize);
  FStream := AStream;
  FOwnsStream := AOwnsStream;
end;

destructor TChunkParser.Destroy;
begin
  if FOwnsStream then FreeAndNil(FStream);
  inherited;
end;

function TChunkParser.GetCurrent: TBytes;
begin
  Result := FBuffer;
end;

function TChunkParser.GetEnumerator: IChunkParser;
begin
  Result := Self;
end;

function TChunkParser.MoveNext: Boolean;
var
  BytesRead: Integer;
begin
  Result := False;
  if FStream = nil then Exit;
  //NB: TStream itself is covered in chapter 8
  BytesRead := FStream.Read(FBuffer[0], Length(FBuffer));
  Result := (BytesRead > 0);
  if BytesRead <> Length(FBuffer) then
  begin
    SetLength(FBuffer, BytesRead);
    if FOwnsStream then FreeAndNil(FStream);
  end;
end;
```

The `ParseChunks` function itself simply returns a new instance of `TChunkParser`:

```
function ParseChunks(AStream: TStream; AChunkSize: TChunkSize;
  AOwnsStream: Boolean = False): IChunkParser;
begin
  Result := TChunkParser.Create(AStream, AChunkSize, AOwnsStream);
end;

function ParseChunks(const AFileName: string;
  AChunkSize: TChunkSize): IChunkParser;
```

```
begin
  Result := ParseChunks(TFileStream.Create(AFileName,
    fmOpenRead or fmShareDenyWrite), AChunkSize, True)
end;
```

Notice that if the stream is owned, it may be freed in more than one place. This is to cover the cases of when the `for`/`in` loop is both completed and when it is broken off: when it is completed, `MoveNext` will have a chance to return `False`, allowing it to free an owned stream there and then. When the `for`/`in` loop is broken off however (e.g. by use of the `Break` statement, or an exception being raised), freeing an owned stream will have to wait until the enumerator goes completely out of scope, causing its `Destroy` method to be called.

7. Basic I/O: console programs and working with the file system

This chapter looks at the mechanics of writing a program with a command-line interface in Delphi — a 'console' application in other words — together with the tools provided for working with the file system (moving and copying files, manipulating file names and paths, getting and setting file attributes, and so on).

Console I/O

A 'console' application ('command-line tool' in contemporary Apple parlance) is a text-mode program that uses a Command Prompt or Terminal window for its user interface. While you aren't going to write the next App Store sensation with this style of application, it remains a useful one for simple utilities that require minimal or no user interaction, especially when the ability to be called from a batch file or shell script is desirable.

Console vs. GUI applications

While the user experience of console and GUI programs is very different, they have very few technical differences at the level of bits and bytes. On Windows, the file format for a console EXE is identical to a GUI one, with the exception of a single bit set in the header. This instructs the operating system to prepare a console window for the application, however whether the program actually uses it is up to the program. On OS X, there isn't even that difference — what makes a native GUI application a native GUI application is the fact the executable has been put inside a special folder structure (the application 'bundle'). When that is the case, so-called 'standard input' and 'standard output' are still available — it's just there won't be a Terminal window shown by default.

Creating a console application

In the Delphi IDE, you can create a new console application by choosing `File|New|Other...` from the main menu bar and selecting the corresponding option from the `Delphi Projects` node. When you do this, a new project will be created containing no units and just a rather blank-looking DPR file. The only really important thing in it will be the `{$APPTYPE CONSOLE}` line near the top. When targeting Windows, this does three things:

- Has the compiler set the 'make me a console window' bit in the EXE header mentioned previously.
- Causes the `IsConsole` global variable to return `True`.
- Makes the `ReadLn`, `Write` and `WriteLn` standard procedures for reading and writing to the console work without runtime error:

```
program Project1;

{$APPTYPE CONSOLE}

uses
  System.SysUtils, Vcl.Dialogs;

begin
  if IsConsole then
  begin
    WriteLn('This has been compiled as a console application');
    Write('Press ENTER to exit...');
    ReadLn;
  end
  else
    ShowMessage('This has been compiled as a GUI application');
end.
```

When `{$APPTYPE CONSOLE}` isn't set (and not being set is the default), `IsConsole` will return `False` and naïve calls to `ReadLn`, `Write` and `WriteLn` will cause runtime exceptions.

Due to the fact an OS X application can legitimately always write to the console, notwithstanding the fact a Terminal window may not be shown, `{$APPTYPE CONSOLE}` has no effect when targeting the Mac. Instead, `IsConsole` *always* returns `True` and the console I/O routines *always* work. This can be useful for debugging purposes, since when running a program from the IDE, `Write` and `WriteLn` output will appear in the terminal window hosting the remote debugger (alias the 'Platform Assistant Server' — PAServer).

Reading and writing to the console

The way you read and write to the console in Delphi uses functionality that has been around in Pascal and Pascal-based languages since the beginning, namely the `ReadLn`, `Write` and `WriteLn` standard routines. In Delphi's predecessor (Turbo Pascal), they were extended to serve as the way to read and write text files too. While Delphi still supports this, it has better ways to do text file I/O in the form of the `TStreamReader`, `TStreamWriter` and even `TStringList` classes. Consequently, `ReadLn` and company will only be looked at here in their limited console role, not their wider historical one.

Write and WriteLn

`Write` and `WriteLn` output one or more values, with `WriteLn` sending an end-of-line marker as well. The arguments passed can have any numeric, character, boolean or string type — since `Write` and `WriteLn` are provided by the compiler, they do not have to follow ordinary parameter rules):

```
const
  BoolValue: Boolean = True;
  CharValue: Char = 'Z';
  IntValue: Integer = 42;
  RealValue: Real = 123.456;
  StrValue: string = 'this is some text';
begin
  Write('Values: ', BoolValue, ', ', CharValue, ', ',
    IntValue, ', ', RealValue, ', ', StrValue);
end.
```

To control how the output is formatted, you can either use the string formatting functions we met earlier (`Format`, `FloatToStr`, etc.) or a special syntax built into `Write` and `WriteLn` themselves. This syntax allows specifying a minimum character width. Do this by appending a colon followed by the required width immediately after the value itself:

```
WriteLn(IntValue:4);
```

If the length of the value's string representation is less than the minimum width, it is padded to the left with spaces. In the case of floating point values, you can also specify the number of decimal places to output via a second colon delimiter:

```
WriteLn(RealValue:4:2); //min width 4, 2 decimal places
```

If you have reason to use this functionality outside of `Write` and `WriteLn`, you can do so via the `Str` standard routine. This acts like a single-parameter version of `Write` that outputs to a string variable rather than the console:

```
var
  S: string;
begin
  Str(RealValue:4:2, S);
```

In practice, it is usually better to use `Format` rather than `Str` though. This is because the `System.SysUtils` function is both more flexible and more idiomatic in a Delphi context — `Str` is essentially a holdover from Turbo Pascal.

ReadLn

When `ReadLn` is called with no arguments, the program waits for the user to press Enter. When one variable is passed in, `ReadLn` waits for the user to type something *then* press Enter; if more than one variable is passed, it waits for the user to type something, press Enter, type something else and press Enter again, and so on, until each variable has been filled in:

```
var
  First, Second: Integer;
begin
  WriteLn('Type two integers, pressing ENTER after each one:');
  ReadLn(First, Second);
  WriteLn('You typed ', First, ' and ', Second);
  Write('Press ENTER to exit...');
  ReadLn;
end.
```

`ReadLn` accepts any string, numeric, or character variable. If a numeric variable is passed but the user doesn't enter a number, an `EInOutError` exception is raised by default. This is frequently undesirable, given a simple typo can cause it, and typos are hardly an 'exceptional' occurrence.

As a one-off, the compiler therefore supports a retro 'error number' scheme as an alternative: set the `{$I-}` compiler directive, and instead of an exception being raised, the `IOResult` global variable is assigned an error number, 106 in the case of bad numeric input and 0 on success:

```
var
  Value: Integer;
begin
  Write('Enter an integer: ');
  {$I-} //disable raising of EInOutError
  repeat
    Readln(Value);
    if IOResult = 0 then Break;
    Write('Error! Please try again: ');
  until False;
  {$I+} //re-enable raising of EInOutError
  WriteLn('You entered ', Value);
  Write('Press ENTER to exit...');
  ReadLn;
```

```
end.
```

A third alternative is to request a string and do your own conversion. In my view this is by far the best option, since you are able to control what is classed as 'valid' input:

```
uses System.SysUtils; //for Trim and TryStrToInt

var
  S: string;
  Value: Integer;
begin
  Write('Enter another integer: ');
  repeat
    ReadLn(S);
    if TryStrToInt(Trim(S), Value) then
      Break
    else
      Write('Error! Please try again: ');
  until False;
  WriteLn('You entered ', Value);
end.
```

Parsing command line parameters

While they aren't specific to console applications, the RTL's functions for parsing command line parameters are particularly applicable to them. To retrieve the number of parameters passed to the application, call `ParamCount`; to actually retrieve a parameter by its index, use `ParamStr` (both are defined in the `System` unit). `ParamStr` is 1-indexed, with `ParamStr(0)` returning the path to the application itself (Windows) or the command used to start the application (OS X):

```
var
  I: Integer;
begin
  WriteLn('My path is ', ParamStr(0));
  Writeln(ParamCount, ' parameters passed to me');
  for I := 1 to ParamCount do
    Writeln(ParamStr(I));
```

These functions assume standard command line semantics. Thus, parameters are delimited with spaces, and parameters that contain one or more spaces are wrapped with "double quote" characters.

Layered on top of `ParamStr` and `ParamCount` is the `FindCmdLineSwitch` function, which is provided by the `System.SysUtils` unit. This function allows you to easily locate individual switches passed on the command line. Slightly simplifying, the declarations for it look like this:

```
const
  {$IFDEF MSWINDOWS}
  SwitchChars = ['/','-']
  {$ELSE}
  SwitchChars = ['-']
  {$ENDIF};

type
  TCmdLineSwitchTypes = set of (clstValueNextParam, clstValueAppended);

function FindCmdLineSwitch(const Switch: string;
  const Chars: TSysCharSet = SwitchChars;
    IgnoreCase: Boolean = True): Boolean; overload;

function FindCmdLineSwitch(const Switch: string;
  var Value: string; IgnoreCase: Boolean = True;
  const SwitchTypes: TCmdLineSwitchTypes =
    [clstValueNextParam, clstValueAppended]): Boolean;
```

On Windows, either a forward slash or a hyphen denote the start of a switch by default, otherwise just a hyphen does, given the forward slash is the path delimiter on *nix platforms.

Here's `FindCmdLineSwitch` in action:

```
uses
  System.SysUtils;

var
  LogFileName: string;
  QuickScan, UseLogFile, Verbose: Boolean;
begin
  if FindCmdLineSwitch('LogFile', LogFileName) then
```

```
    Writeln('Log file: ', LogFileName)
  else
    Writeln('No log file');
  QuickScan := FindCmdLineSwitch('QuickScan');
  Verbose := FindCmdLineSwitch('Verbose');
  WriteLn('Quick scan? ', QuickScan);
  WriteLn('Verbose? ', Verbose);
end.
```

For testing purposes, you can set an application's command line parameters in the IDE via `the Run|Parameters...` menu command. Say you did that for the program just listed and entered the following:

```
-quickscan -logfile:C:\Users\CCR\MyApp.log
```

The program would then output this:

```
Log file: C:\Users\CCR\MyApp.log
Quick scan? TRUE
Verbose? FALSE
```

By default, either of the following command lines would produce the same result:

```
-quickscan -logfileC:\Users\CCR\MyApp.log
-quickscan -logfile C:\Users\CCR\MyApp.log
```

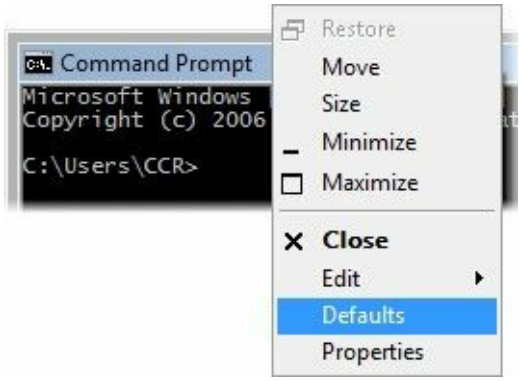
While allowing the value to come immediately after the switch can't be changed so long as you allow a colon in between, explicitly passing an argument for the `SwitchTypes` parameter can either enforce or disallow having a space in the middle — use `[clstValueNextParam]` to enforce it and `[clstValueAppended]` to disallow it. Nonetheless, the `clstValueNextParam` style is not required to cover the case of a value that contains a space, since the colon style handles that just fine:

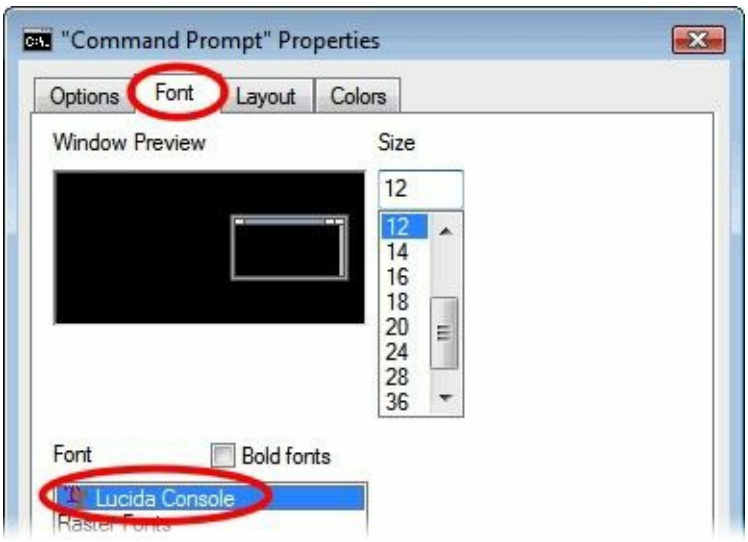
```
-quickscan -logfile:"C:\Users\CCR\My App.log"
```

Console I/O and Unicode

Unfortunately, Unicode and the Windows Command Prompt do not get along very well, since by default, output is restricted to the legacy 'Ansi' codepage, if that. Given this codepage is Latin-1 ('Western European') on my computer not Cyrillic, calling `WriteLn('Γειά σου Κόσμε')` outputs `Ge?? s?? ??sμe` for me.

Doing better than this is possible however. Step one is to ensure a Unicode-supporting font like Lucida Console or Consolas is used rather than the default 'Raster Fonts'. This is best done globally: with a Command Prompt open, right click on the title bar and select Defaults, followed by the Fonts tab:





Once a suitable font is set, the next step is for the program itself to notify Windows that Unicode (specifically, UTF-8) will be outputted, and the Delphi RTL told to actually output it. This is done by calling the `SetConsoleOutputCP` API function and `SetTextCodePage` RTL procedure respectively:

```
uses
  Winapi.Windows;

begin
  SetConsoleOutputCP(CP_UTF8);
  SetTextCodePage(Output, CP_UTF8);
  WriteLn('Γειά σου Κόσμε');
  ReadLn;
end.
```

Unfortunately, even once these steps have been taken, it is unlikely full Unicode support will have been achieved. In particular, while outputting (say) Greek characters will now work, showing Arabic or Chinese text will probably still fail. This is due to limitations surrounding the available fonts (Lucida Console and Consolas). Happily, Terminal on the Macintosh has no such problems: it correctly outputs Unicode characters without you needing to do anything.

Console vs. form-based applications?

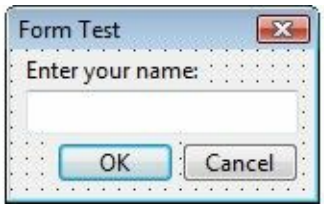
On Windows, a GUI application can choose to allocate a console explicitly by calling the `AllocConsole` API function:

```
{$APPTYPE GUI} //This is the default, but make it explicit
               //for demonstration purposes
uses Winapi.Windows, System.SysUtils;

begin
  if not AllocConsole then RaiseLastOSError;
  WriteLn('IsConsole=', IsConsole); //still False
  WriteLn('We have a console window to write to!');
  ReadLn;
end.
```

Conversely, compiling as a console application does not prevent you from using forms and other GUI components. Indeed, you could take a normal VCL or FireMonkey application, set the `{$APPTYPE CONSOLE}` directive, and have it become a console program in formal terms.

Much more common will be the case of wanting to use the odd dialog box in a console application though. When using the VCL, this is a perfectly valid aim. Let's run through an example.

To begin, create a new console application in the IDE, and add a new VCL form to it. In the form designer, place a `TLabeledEdit` and two buttons on the form, and set them up to look like the following:



In the Object Inspector, rename the `TLabeledEdit` component `edtName`, set the OK button's `ModalResult` property to `mrOK`, and the cancel button's `ModalResult` property to `mrCancel`; with the form itself, set its `BorderStyle` property to `bsDialog`, its `Name` property to `frmNameDlg`, and its `Position` property to `poScreenCenter`. Then, press `F12` to go into the code editor, and add the following global function:

```
function ShowNameDialog(out EnteredName: string): Boolean;
var
  Form: TfrmNameDlg;
begin
  Form := TfrmNameDlg.Create(nil);
  try
    Result := IsPositiveResult(Form.ShowModal);
    if Result then EnteredName := Form.edtName.Text;
  finally
    Form.Free;
  end;
end;
```

After declaring this in the interface section of the unit, go back to the project's source (the DPR) and amend the uses clause downwards to look like this:

```
uses
```

```
  Vcl.Forms,
  Vcl.Dialogs,
  Unit1 in 'Unit1.pas' {frmNameDlg};

var
  Dlg: TOpenDialog;
  EnteredName: string;
begin
  Application.Title := 'Console with VCL Usage Test';
  { Show a common dialog box... }
  Dlg := TOpenDialog.Create(nil);
  try
    Dlg.DefaultExt := 'txt';
    Dlg.Filter := 'Text files (*.txt)|*.txt|All files (*.*)|*.*';
    Dlg.Options := [ofFileMustExist];
    if Dlg.Execute then
      MessageDlg('You chose "' + Dlg.FileName + '"',
        mtInformation, [mbOK], 0)
    else
      MessageDlg('You cancelled!', mtWarning, [mbOK], 0);
  finally
    Dlg.Free;
  end;
  { Show a VCL form... }
  if ShowNameDialog(EnteredName) then
    ShowMessage('Hello ' + EnteredName)
  else
    ShowMessage('Oi - you cancelled my lovely form!');
end.
```

Run the program, and you should find yourself successfully using both a standard ‘file open’ dialog and a custom `TForm` descendant.

While it is easy enough to do, there are a couple of small ‘gotchas’ when mixing forms and dialog boxes with console applications on Windows. The first is that the IDE will not generate a ‘manifest’ for a console project by default, which means GUI won’t be shown themed, and instead have the Windows 2000 ‘look’. Adding `Vcl.XPMan` to the DPR’s uses clause is a quick way of fixing that however:

```
uses
  Vcl.Forms,
  Vcl.Dialogs,
  Vcl.XPMan,   //!!! added
  Unit1 in 'Unit1.pas' {frmNameDlg};
```

The second issue is that COM (the Component Object Model) must be initialised if open or save dialog boxes are used, due to the fact Windows Explorer (which the file dialogs use) can host COM-based plugins. If you include at least one VCL form in the project initialising COM will be done automatically. Otherwise, you should add the `System.Win.ComObj` unit to the DPR’s uses clause:

```
uses
  System.Win.ComObj,  //!!! added
  Vcl.Dialogs,
  Vcl.XPMan;
```

Doing this will ensure COM is loaded property.

Working with the file system

Routines for working with the file system are spread over two units, `System.SysUtils` and `System.IOUtils`. Where the first provides a set of standalone functions, the second is based around three different record types — `TFile`, `TDirectory` and `TPath` — and the static methods they contain. The functionality of the two units partially overlaps; as we go through them, advice will therefore be given about the better alternative when more than one option is available for a given purpose.

Testing for a file's existence

To test whether a given file exists, pass the file name to either `FileExists` (`System.SysUtils`) or `TFile.Exists` (`System.IOUtils`). Since `TFile.Exists` does nothing more than call `FileExists`, these functions are strictly equivalent:

```
if FileExists('C:\Test\Foo.txt') then ...
if TFile.Exists('C:\Test\Foo.txt') then ...
```

If the path passed in is to a symbolic link (symlink), the link is followed through by default — pass `False` as a second argument to prevent this:

```
if FileExists(SomePath, False) then ...
```

Be warned that symbolic links are not the same things as shortcut files on Windows — if passed the path to a shortcut (*.LNK), only the existence of the shortcut file is tested for.

The path passed to `FileExists` or `TFile.Exists` can be either absolute (e.g. `'C:\Users\CCR\Documents\Test.doc'`) or relative (e.g. `'Documents\Test.doc'`, `'Test.doc'`, `'..\Test.doc'`, etc.). If a relative path is specified, the application's current working directory is used to create an absolute one internally. Especially in a GUI application, the current directory can easily be changed behind the application's back (e.g., by a file open or save dialog). Because of this, you should always use absolute paths when calling `FileExists`, or indeed any other file system routine.

For that purpose, the `ExpandFileName` function is available to easily convert a relative to an absolute path explicitly:

```
AbsPath := ExpandFileName('Foo.txt');
```

On OS X, this function will also need to be used when a path string uses the `~/` shortcut to denote the user's 'home' directory:

```
AbsPath := ExpandFileName('~/Documents/Foo.txt');
```

Fail to do this, and you will get 'file does not exist' errors when attempting to read files, e.g. when calling the `LoadFromFile` method of a `TBitmap` or `TStrings` object.

Renaming, deleting and moving files

To rename a file, call `RenameFile`; to delete one, call `DeleteFile`. Both are functions that return `True` on success, `False` on failure:

```
if RenameFile('C:\Foo\Old.txt', 'C:\Foo\New.txt') then
  WriteLn('Success!')
else
  WriteLn('Could not rename file');
```

Depending on the circumstances, `RenameFile` may also be able to move a file from one location to another. However, the proper way to do this is to call `TFile.Move`:

```
TFile.Move('C:\Old place\Foo.txt', 'C:\New place\Foo.txt');
```

Unlike `RenameFile` and `DeleteFile`, `TFile.Move` is a procedure that will raise an exception on failure.

`TFile` also has a `Delete` method. However, using `DeleteFile` is the better option since `TFile.Delete` is a procedure that only raises an exception indicating failure if the reason was a sharing violation (in other words, if another program currently has the file open). In contrast, `DeleteFile` is a function that will return `False` on failure for whatever reason.

Retrieving and changing file dates and times

`TFile` contains methods for getting and setting a file's various time stamps (creation, last access and last written to), in both local and UTC/GMT formats:

```
procedure OutputDateTimes(const AFileName: string);
begin
  WriteLn('Creation time: ',
    DateTimeToStr(TFile.GetCreationTime(AFileName));
  WriteLn('Last access time: ',
    DateTimeToStr(TFile.GetLastAccessTime(AFileName));
```

```
  WriteLn('Last write time: ',
    DateTimeToStr(TFile.GetLastWriteTime(AFileName));
  WriteLn('Last write time (GMT): ',
    DateTimeToStr(TFile.GetLastWriteTimeUtc(AFileName));
  TFile.SetLastAccessTime(AFileName, Now);
end;
```

Internally, the `GetXXX` methods boil down to the `FileGetDateTimeInfo` standalone function:

```
procedure OutputDateTimes2(const AFileName: string);
var
  Rec: TDateTimeInfoRec;
begin
  if not FileGetDateTimeInfo(AFileName, Rec) then
  begin
    WriteLn('File appears not to exist!');
    Exit;
  end;
  WriteLn('Creation time: ', DateTimeToStr(Rec.CreationTime);
  WriteLn('Last access time: ', DateTimeToStr(Rec.LastAccessTime);
  WriteLn('Last write time: ', DateTimeToStr(Rec.TimeStamp);
end;
```

`System.SysUtils` also provides an overloaded `FileAge` routine — use the `TDateTime` version, if any. `FileAge` is equivalent to `TFile.GetLastWriteTime`, but on Windows is more robust because it will work even if the file in question is currently open for exclusive access by another application. In contrast, `TFile.GetLastWriteTime` is liable not to.

Getting file attributes

To discover whether operating system considers a file read only, call `FileIsReadOnly`. This function returns `True` if the file both exists and is read only and `False` otherwise.

To read other sorts of file attribute, call either `FileGetAttr` from `System.SysUtils` or `TFile.GetAttributes` from `System.IOUtils`. These functions have a couple of differences. The first is that `FileGetAttr` is weakly typed, working with `faXXX` bitset flags: `faReadOnly`, `faHidden`, `faSysFile`, `faDirectory`, `faArchive`, `faNormal`, `faTemporary`, `faSymLink`, `faCompressed`, `faEncrypted`, `faVirtual`. These directly map to Windows file attributes, so that `faReadOnly` has the same value as the API-defined `FILE_ATTRIBUTE_READONLY` constant, for example. Being weakly typed means you test for a specific `faXXX` value using the `and` operator:

```
var
  Attr: Integer;
begin
  Attr := FileGetAttr(SomePath);
  if faHidden and Attr <> 0 then
    WriteLn('We have a hidden file or directory!');
  if faDirectory and Attr <> 0 then
    WriteLn('We have a directory!');
```

In contrast, `TFile.GetAttributes` is strongly typed, working with a proper set type. While the element names are similarly named to their `SysUtils` equivalents, there is no clash due to a scoped enumeration being used:

```
var
  Attr: TFileAttributes;
begin
  Attr := TFile.GetAttributes(SomePath);
  if TFileAttribute.faHidden in Attr then
    WriteLn('We have a hidden file or directory!');
  if TFileAttribute.faDirectory in Attr then
    WriteLn('We have a directory!');
```

The second difference between `FileGetAttr` and `TFile.GetAttributes` is how they deal with Windows and OS X differences, since the range of possible file attributes between the two platforms is quite different. `FileGetAttr` takes the approach of mapping the core Windows attributes — specifically, `faReadOnly`, `faDirectory`, `faHidden` and `faSymLink` — to *nix equivalents, with all the others being treated as Windows-specific (in the case of `faHidden`, this means returning it when the file or directory begins with a full stop, which conventionally denotes a 'hidden' file system entry on *nix). In contrast, the `TFileAttributes` set type used by `TFile.GetAttributes` has a different range of possible values depending on the target platform. On Windows the base enumerated type, `TFileAttribute`, is declared like this:

```
TFileAttribute = (faReadOnly, faHidden, faSystem,
  faDirectory, faArchive, faDevice, faNormal, faTemporary,
  faSparseFile, faReparsePoint, faCompressed, faOffline,
  faNotContentIndexed, faEncrypted, faSymLink);
```

On OS X it becomes this though:

```
TFileAttribute = (faNamedPipe, faCharacterDevice,
  faDirectory, faBlockDevice, faNormal, faSymLink, faSocket,
  faWhiteout, faOwnerRead, faOwnerWrite, faOwnerExecute,
  faGroupRead, faGroupWrite, faGroupExecute, faOthersRead,
  faOthersWrite, faOthersExecute, faUserIDExecution,
  faGroupIDExecution, faStickyBit);
```

In both cases, the elements map to the possible file system attributes native to the platform.

Setting file attributes

To change a file's read only state, call `FileSetReadOnly`. This is a function that returns `True` on success and `False` on failure, for example because the specified file does not exist. On Windows it attempts to change the 'read only' file attribute; on OS X, it attempts to set or remove the user, group and 'other' write permissions on the file.

As `FileGetAttr` and `TFile.GetAttributes` read file attributes, so `FileSetAttr` and `TFile.SetAttributes` write them. If you want to set attributes and target OS X, your choice is made for you however because `FileSetAttr` only exists on Windows.

In either case you will probably want to assign a modified version of what were the attributes for a given file. For example, on Windows, a function like `FileSetReadOnly` that sets the hidden attribute rather than the read-only one should be written like this:

```
function FileSetHidden(const AFileName: string;
  ANewHiddenState: Boolean): Boolean;
var
  Attr: Integer;
begin
  Attr := FileGetAttr(AFileName);
  if Attr = faInvalid then Exit(False);
  if ANewHiddenState then
    Attr := Attr or faHidden
  else
    Attr := Attr and not faHidden;
  Result := (FileSetAttr(AFileName, Attr) = 0);
end;
```

FileSetAttr issues

Aside from being Windows-only, `FileSetAttr` also suffers from limitations stemming from being a very thin wrapper over the `SetFileAttributes` API function. In particular, it is incapable of setting `faEncrypted` or `faCompressed`.

In the case of setting or removing the `faEncrypted` attribute though, the workaround is easy: call `TFile.Encrypt` or `TFile.Decrypt` as appropriate. Changing a file's compressed status is a little trickier, since a small bit of API coding is necessary:

```
uses Winapi.Windows;

function FileSetCompressed(const FileName: string;
  Compress: Boolean): Boolean;
var
  Handle: THandle;
  NewValue: USHORT;
  Dummy: DWORD;
begin
  Handle := CreateFile(PChar(FileName), GENERIC_READ or
    GENERIC_WRITE, FILE_SHARE_READ, nil, OPEN_EXISTING,
    FILE_FLAG_BACKUP_SEMANTICS, 0);
  if Handle = INVALID_HANDLE_VALUE then Exit(False);
  try
    NewValue := Ord(Compress);
    Result := DeviceIoControl(Handle, FSCTL_SET_COMPRESSION,
      @NewValue, SizeOf(NewValue), nil, 0, Dummy, nil);
  finally
    CloseHandle(Handle);
  end;
end;
```

Determining whether the file system — and more specifically, that of the 'volume' to be written to — supports either encrypted or compressed attributes in the first place requires another API call:

```
uses Winapi.Windows, System.SysUtils;

function GetFileSystemFlags(const FileName: string): DWORD;
var
  Dr: string;
```

```
begin
  Dr := IncludeTrailingPathDelimiter(ExtractFileDrive(FileName));
  if not GetVolumeInformation(PChar(Dr), nil, 0, nil,
    PDWORD(nil)^, Result, nil, 0) then RaiseLastOSError;
end;

function CanUseCompressedAttr(const FileName: string): Boolean;
begin
  Result := GetFileSystemFlags(FileName) and
    FS_FILE_COMPRESSION <> 0;
end;

function CanUseEncryptedAttr(const FileName: string): Boolean;
begin
  Result := GetFileSystemFlags(FileName) and
    FILE_SUPPORTS_ENCRYPTION <> 0;
end;
```

Listing and searching for files and directories

The simplest way to retrieve all files in a given directory is to call `TDirectory.GetFiles`. This returns a dynamic array of file name strings, and has a number of overloads. While the most basic just takes a path, returning all files in the immediate directory, others allow you to specify a search pattern (e.g. `*.doc`), whether to search sub-directories too, and a filtering callback method. Here's an example that specifies all three:

```
var
  FileName: string;
begin
  for FileName in TDirectory.GetFiles('C:\Users\CCR\Documents',
    '*.rtf', TSearchOption.soAllDirectories,
    function (const Path: string; const Rec: TSearchRec): Boolean
    begin
      Result := (Rec.Size > 102400); //over 100KB
    end) do WriteLn(FileName);
  ReadLn;
end.
```

Here, `'C:\Users\CCR\Documents'` is the folder to search, `'*.rtf'` the search pattern (namely, to find all files with an RTF extension), `TSearchOption.soAllDirectories` requests to search sub-directories too, and the anonymous method the callback that tests for whether the size is over 102,400 bytes (= 100 KB).

Internally, `TDirectory.GetFiles` wraps the `FindFirst`/`FindNext`/`FindClose` functions declared by `System.SysUtils`. Unless your needs are very simple, using the `FindXXX` functions directly is likely to be the better option. This is because calling `TDirectory.GetFiles` means the whole search path is enumerated before the function returns; in contrast, using `FindXXX` directly means you can break off the search midway.

Here's the basic usage pattern:

```
var
  SearchRec: TSearchRec;
begin
  if FindFirst('C:\*.doc', faAnyFile and not faDirectory,
    SearchRec) = 0 then
  try
    repeat
      //work with SearchRec
    until FindNext(SearchRec) <> 0;
  finally
    FindClose(SearchRec);
  end;
```

The first parameter to `FindFirst` takes a path specification, which may include wildcards; the second takes a file attribute mask, which is a combination of the `faXXX` constants we met earlier; and the third takes a `TSearchRec` record, which is filled with information about the found file.

On all platforms, `TSearchRec` contains the following members:

```
Size: Int64;
Attr: Integer;
Name: TFileName;
ExcludeAttr: Integer;
property TimeStamp: TDateTime read GetTimeStamp;
```

Here, `Size` is the file's size in bytes; `Attr` a bitmask of `faXXX` values that pertain to the file; `Name` *just* the name part of the file name (e.g. `MyFile.txt` not `C:\Stuff\MyFile.txt`); `ExcludeAttr` a bitmask that includes those of `faHidden`, `faSysFile` and

`faDirectory` that the second parameter to `FindFirst` implicitly excluded (more on which shortly); and `TimeStamp` the date/time the file was last written to, as reported by the OS.

Going deeper with FindFirst, FindNext and FindClose

One potential gotcha of `FindFirst` is that the path specification is passed 'as is' to the underlying operating system API. On Windows, this can lead to unintended results due to the fact every 'long' file name is given a corresponding 'short' (8.3) file name too. For example, if you search for `'*.doc'`, then files ending with the `docx` or `docm` extensions will be returned as well as `doc` ones. This is standard Windows behaviour however — the same thing happens if you enter `DIR *.doc` at a Command Prompt, for example.

The solution to the problem (if you consider it a problem) is to search for any file in the first instance before doing some sort of 'second level' filtering yourself. The `MatchesMask` function of the `System.Masks` unit is ideal for this task:

```
uses System.SysUtils, System.Masks;

var
  Rec: TSearchRec;
begin
  if FindFirst('C:\*', 0, Rec) = 0 then
  try
    repeat
      if MatchesMask(Rec.Name, '*.doc') then
      begin
        WriteLn(Rec.Name);
      end;
    until FindNext(Rec) <> 0;
  finally
    FindClose(Rec);
  end;
```

Another potentially confusing thing about `FindFirst` is its second parameter. In it, you specify what sorts of file system item to return, using the `faXXX` constants we previously met when looking at `FileGetAttr` and `FileSetAttr`. Only the following combinations of the `faXXX` constants actually have any effect however:

- `faAnyFile` alone, which in fact means any file *or sub-directory*.
- Any combination of `faDirectory`, `faHidden` and `faSysFile`. This will mean: return any normal file *plus* any file system entry that has one or more of the specified 'special' attributes. To specify more than one, use the logical OR operator, e.g. `faDirectory or faHidden`.
- `faAnyFile` with any of `faDirectory`, `faHidden` and/or `faSysFile` taken out (e.g. `faAnyFile and not faDirectory`).

If 0 or any other `faXXX` constant is passed, only files that are *not* hidden or system files are returned. This makes passing 0 functionally equivalent to passing `faAnyFile and not (faDirectory or faHidden or faSysFile)`.

In themselves, the `FindXXX` functions only search a single directory. To search sub-directories too, some simple recursion needs to be employed. Here's a generalised implementation of the pattern:

```
type
  TEnumFilesCallback = reference to procedure
    (const AName: string; const AInfo: TSearchRec;
     var AContinueSearching: Boolean);

procedure EnumFilesRecursively(const BasePath: string;
  const Callback: TEnumFilesCallback);

  function SearchPath(const DirPath: string): Boolean;
  var
    Info: TSearchRec;
  begin
    Result := True;
    if FindFirst(DirPath + '*', faAnyFile, Info) <> 0 then Exit;
    try
      repeat
        if Info.Attr and faDirectory = 0 then
          Callback(DirPath + Info.Name, Info, Result)
        else if (Info.Name <> '.') and (Info.Name <> '..') then
          Result := SearchPath(DirPath + Info.Name + PathDelim);
      until not Result or (FindNext(Info) <> 0);
    finally
      FindClose(Info);
    end;
```

```
  end;
begin
  SearchPath(IncludeTrailingPathDelimiter(BasePath));
end;
```

Notice the check for the `..` and `.` pseudo-directories — since `..` means 'the parent directory' and `.` 'the current directory', not filtering them out will lead to an endless loop. That said, `EnumFilesRecursively` can be used like this:

```
  EnumFilesRecursively('C:\Misc\Stuff',
    procedure (const FullName: string; const Info: TSearchRec;
      var ContinueSearching: Boolean)
    begin
      WriteLn(FullName);
    end);
```

Adapt the input path accordingly, and the same routine will work without any changes on OS X too.

Working with directories and drives

Checking whether a directory exists

Similar to testing for a file's existence with `FileExists` or `TFile.Exists`, you can test for a directory's existence by calling either `DirectoryExists` from `System.SysUtils` or `TDirectory.Exists` from `System.IOUtils`. As before, the `IOUtils` version simply delegates to the `SysUtils` one:

```
if DirectoryExists('C:\Test') then ...
if TDirectory.Exists('C:\Test') then ...
```

Retrieving sub-directory names

Similar to the `GetFiles` method mentioned previously, `TDirectory` also has a `GetDirectories` function, which returns a dynamic array of strings that contain the names of the specified directory's sub-directories. As with `GetFiles`, this function is implemented by calling `FindFirst`/`FindNext`/`FindClose`.

Getting and setting the current working directory

To get the current working directory, call either `GetCurrentDir` from `System.SysUtils` or `TPath.GetCurrentDirectory`; to set it, call either `SetCurrentDir` or `TPath.SetCurrentDirectory`. While the first pair are directly equivalent, the second differ on failure: where `SetCurrentDir` (being a `Boolean` function) returns `False`, `TPath.SetCurrentDirectory` (a procedure) raises an exception.

Creating directories

For creating directories, `System.SysUtils` provides `CreateDir` and `ForceDirectories` functions; over in `System.IOUtils`, `TDirectory` has a `CreateDirectory` method. All `TDirectory.CreateDirectory` does is pass on the request to `ForceDirectories` though. The difference between `CreateDir` and `ForceDirectories` is that `CreateDir` can only create a sub-directory for a parent that already exists. In contrast, `ForceDirectories` will create a whole folder hierarchy if necessary. For example, if there is no `C:\Test`, then `CreateDir('C:\Test\Foo')` will fail where `ForceDirectories('C:\Test\Foo')` will succeed, assuming the application has the rights to do so from the operating system.

Moving and removing directories

Delete a directory by calling `TDirectory.Delete`, passing `True` as the second parameter to remove all files and subdirectories too. While `System.SysUtils` has a `RemoveDir` function, this only works if the directory is empty. Moreover, on Windows, `TDirectory.Delete` will clear any deletion-preventing attributes on a directory first before attempting to remove it, which `RemoveDir` doesn't do.

To move a directory, call `TDirectory.Move`:

```
TDirectory.Move('C:\Foo', 'C:\Bar');
```

Here, first `C:\Bar` is created if it doesn't already exist, the contents of `C:\Foo` moved over, before `C:\Foo` itself is deleted.

Retrieving special directories

The path to the current executable can be retrieved by calling `GetModuleName(0)`. On Windows, you can also use `ParamStr(0)`:

```
  WriteLn(ExtractFilePath(GetModuleName(0)));
  WriteLn(ExtractFilePath(ParamStr(0)));
```

When used in a FireMonkey application compiled for OS X, you should be clear about what you are getting. This is because a FireMonkey executable on OS X, being a graphical OS X application, will be buried inside a special folder structure (the 'bundle'). For example, say your program is called `SuperProg`; as deployed, this will involve a folder called `SuperProg.app` (the `.app` extension will be hidden in Finder), with the executable itself placed under `SuperProg.app/Contents/MacOS/SuperProg` (OS X executables have no extension). In contrast, a compiled Windows program intrinsically involves nothing more than the executable — the EXE — itself.

For the purpose of finding a suitable directory for program settings, you can call either `TPath.GetHomePath` or the `GetHomePath` function in `System.SysUtils`. On Windows, this returns the path to the roaming Application Data folder:

```
  WriteLn(TPath.GetHomePath); //e.g. C:\Users\CCR\AppData\Roaming
```

In practice, you'll probably want to call `GetHomePath` along with `ForceDirectories`:

```
function BuildSettingsPath(const CompanyName,
```

```
  ProductName: string): string;
begin
  Result := IncludeTrailingPathDelimiter(GetHomePath) +
    CompanyName + PathDelim + ProductName + PathDelim;
  ForceDirectories(Result);
end;
```

Rather lamely, on OS X `GetHomePath` just returns the home directory. Thus, if the function returns `C:\Users\BobSmith\AppData\Roaming` on Windows, it will return only `/users/BobSmith` on OS X, despite `/users/BobSmith/Library` being the equivalent folder. To do things properly you therefore have to drop down to the operating system API:

```
uses
  Macapi.CoreFoundation, Macapi.ObjectiveC, Macapi.Foundation;

function GetMacLibraryDir: string;
var
  Manager: NSFileManager;
  URL: NSURL;
  Range: CFRange;
begin
  //get the path into a NSURL object
  Manager := TNSFileManager.Wrap(TNSFileManager.OCClass.defaultManager);
  URL := Manager.URLForDirectory(NSLibraryDirectory,
    NSUserDomainMask, nil, False, nil);
  //convert the NSURL into a Delphi string
  Range.location := 0;
  Range.length := URL.path.length;
  SetLength(Result, Range.length);
  CFStringGetCharacters((URL.path as ILocalObject).GetObjectID,
    Range, PChar(Result));
end;
```

Aside from the user's 'home' directory, `TPath` also allows you to retrieve their TEMP directory:

```
S:= TPath.GetTempPath; //e.g. C:\Users\CCR\AppData\Local\Temp\
```

On a Mac this will return a funny-looking path, however that is nothing to worry about — unlike Windows, OS X creates per-process temporary directories on the fly.

To retrieve any other special directories, you will need to either call a relevant operating system API function or (if applicable) retrieve a relevant environment variable using the `GetEnvironmentVariable` function defined by `System.SysUtils`. Here's an example of the second approach (if the specified variable doesn't exist, an empty string is returned):

```
function GetCommonFilesDir: string;
begin
  Result := GetEnvironmentVariable('CommonProgramFiles');
end;
```

On Windows, the most popular API to do things 'properly' is probably `SHGetFolderPath`, a function declared in the `System.ShlObj` unit in Delphi. Here's an example that retrieves the Common Files directory as before, only with appropriate exceptions being raised in the (albeit unlikely) event of failure:

```
uses
  Winapi.Windows{ for MAX_PATH }, Winapi.ShlObj,
  System.RTLConsts{ for SPathNotFound }, System.SysUtils;

function GetCommonFilesDir: string;
var
  Buffer: array[0..MAX_PATH] of Char;
begin
  case SHGetFolderPath(0, CSIDL_PROGRAM_FILES_COMMON, 0,
        SHGFP_TYPE_CURRENT, Buffer) of
    S_OK: Exit(Buffer);
    E_FAIL: raise EDirectoryNotFoundException.CreateRes(@SPathNotFound);
  else raise EArgumentException.CreateRes(@SArgumentOutOfRange);
  end;
end;
```

Getting logical drive and disk information

To get a list of logical drives on the file system, call `TDirectory.GetLogicalDrives`. This returns a dynamic array of drive strings:

```
var
  Drive: string;
```

```
begin
  for Drive in TDirectory.GetLogicalDrives do
    WriteLn(Drive);
```

Output would look something like this, assuming a desktop Windows program:

```
C:\
D:\
E:\
F:\
```

Since OS X does not have the concept of drive letters, an empty array is always returned on that platform.

To extract the drive part from a file name, you can use either `ExtractFileDrive` from `System.SysUtils` or `TPath.GetPathRoot`. Their semantics are slightly different, as the following example shows:

```
uses System.SysUtils, System.IOUtils;

const
  DesktopPath = 'C:\Users\Public\Documents\Policy.doc';
  NetworkPath = '\\server\share\Documents\Policy.doc';
  UnixPath = '/Public Documents/Policy.doc';
var
  S: string;
begin
  S := ExtractFileDrive(DesktopPath);  { 'C:' }
  S := ExtractFileDrive(NetworkPath);  { '\\server\share' }
  S := ExtractFileDrive(UnixPath);     { '' }

  S := TPath.GetPathRoot(DesktopPath); { 'C:\' }
  S := TPath.GetPathRoot(NetworkPath); { '\\server\share' }
  S := TPath.GetPathRoot(UnixPath);    { '/' }
end.
```

Beyond the differences shown, another one is what happens when an empty string is passed in: where `TPath.GetPathRoot` raises an exception, `ExtractFileDrive` just returns an empty string.

On Windows, a drive's capacity in bytes may be retrieved by calling `DiskSize`; to get the number of bytes still free, call `DiskFree`. Both functions take a numeric identifier rather than a string for input: pass 0 to denote the drive of the current working directory, 1 to denote `A:`, 2 to denote `B:`, 3 to denote `C:`, and so on. Passing an invalid number will cause -1 to be returned. If you want to work with drive letters instead, you could write wrapper functions like the following:

```
uses System.SysUtils;

function DiskCapacity(DriveLetter: Char): Int64;
begin
  case DriveLetter of
    'a'..'z': Result := DiskSize(1 + Ord(DriveLetter) - Ord('a'));
    'A'..'Z': Result := DiskSize(1 + Ord(DriveLetter) - Ord('A'));
  else Result := -1;
  end;
end;

function DiskSpaceFree(DriveLetter: Char): Int64;
begin
  case DriveLetter of
    'a'..'z': Result := DiskFree(1 + Ord(DriveLetter) - Ord('a'));
    'A'..'Z': Result := DiskFree(1 + Ord(DriveLetter) - Ord('A'));
  else Result := -1;
  end;
end;

const
  BytesPerGB = 1024 * 1024 * 1024;
begin
  WriteLn(Format('Size of C: is %.0n GB; %.0n GB is free',
    [DiskCapacity('C') / BytesPerGB, DiskSpaceFree('C') / BytesPerGB]));
end.
```

`DiskFree` can also be used to test whether given drive exists — just test for -1. Alternatively, you can call the `DriveExists` method of `TPath`. This takes neither a numeric identifier nor a single character but a string. This string can contain either just a drive specification or a complete path; if the latter, the file itself need not exist:

```
DriveCExists := TPath.DriveExists('C:\Dummy.txt');
```

Manipulating path and file name strings

Extracting the constituent elements of a path string

Aside from the `ExtractFileDrive` function we have already met, `System.SysUtils` also provides `ExtractFilePath`, `ExtractFileDir`, `ExtractFileName` and `ExtractFileExt` routines:

```
const
  TestFileName = 'C:\Windows\System32\Notepad.exe';
var
  S: string;
begin
  S := ExtractFilePath(TestFileName); //C:\Windows\System32\
  S := ExtractFileDir(TestFileName);  //C:\Windows\System32
  S := ExtractFileName(TestFileName); //Notepad.exe
  S := ExtractFileExt(TestFileName);  //.exe
```

In no case must the file specified actually exist — the functions purely work on the file name string.

Constructing file name strings

To assist constructing path strings, `System.SysUtils` defines `PathDelim` and `DriveDelim` constants. On Windows these have the values of `'\'` and `':'` respectively, otherwise `'/'` and `''` (i.e., an empty string) are used.

To ensure a string has a terminating path delimiter, call `IncludeTrailingPathDelimiter`; to ensure it *doesn't* have one, call `ExcludeTrailingPathDelimiter`. Note these are both functions, leaving untouched the string passed in:

```
const
  P1 = '/users/bobsmith/Documents';
  P2 = '/users/bobsmith/Pictures/';
begin
  WriteLn(IncludeTrailingPathDelimiter(P1));
  WriteLn(ExcludeTrailingPathDelimiter(P1));
  WriteLn(IncludeTrailingPathDelimiter(P2));
  WriteLn(ExcludeTrailingPathDelimiter(P2));
  { output:
  /users/bobsmith/Documents/
  /users/bobsmith/Documents
  /users/bobsmith/Pictures/
  /users/bobsmith/Pictures }
```

To change the file extension in a file name string, call either `ChangeFileExt` from `System.SysUtils` or `TPath.ChangeExtension` from `System.IOUtils`. In either case, actually changing a file's name on disk will require calling `RenameFile` afterwards. `ChangeFileExt` and `TPath.ChangeExtension` have slightly different semantics: most notably, the extension passed to `ChangeFileExt` should contain an initial full stop, otherwise the returned file name string won't have one. In contrast, the `TPath` version explicitly allows for this:

```
const
  FileName = 'Test.doc';
  NewExt = 'rtf';
var
  S: string;
begin
  S := ChangeFileExt(FileName, NewExt);         //Testrtf (sic)
  S := TPath.ChangeExtension(FileName, NewExt); //Test.rtf
```

Allocating names for temporary files

As previously discussed, you can retrieve the path to the current user's (Windows) or current process' (OS X) TEMP directory by calling `TPath.GetTempPath`. For the purpose of allocating a temporary file name, the `TPath` record also provides `GetGUIDFileName`, `GetRandomFileName` and `GetTempFileName` methods. The first of these simply allocates a new GUID, formatting it as one continuous hexadecimal number; in contrast, the second uses the `Random` function to create a gibberish file name:

```
  FN1 := TPath.GetGUIDFileName;
  FN2 := TPath.GetGUIDFileName;
```

Here, `FN1` will be assigned something like `'CA6A46B73C944E77B5C7BBD771A0835E'` and `FN2` something like `'F1547K4k.ssr'`.

Neither `GetRandomFileName` nor `GetRandomFileName` actually test for whether the file name definitely doesn't exist however. In contrast, `TPath.GetTempFileName` does, actually creating an empty file in the TEMP directory in the process:

```
var
```

```
  TempFile: string;
begin
  TempFile := TPath.GetTempFileName;
  try
    //demonstrate we have actually created a file...
    WriteLn('Allocated ' + TempFile + ' at ' +
      DateTimeToStr(TFile.GetCreationTime(TempFile)));
  finally
    DeleteFile(TempFile);
  end;
end.
```

Working with relative paths

To convert an absolute path to a relative one, call `ExtractRelativePath`. The base path specified must have a trailing path delimiter else its final part will be ignored:

```
const
  FN = 'C:\Users\CCR\Documents\File.txt';
begin
  WriteLn(ExtractRelativePath('C:\Users\CCR\', FN));
  WriteLn(ExtractRelativePath('C:\Users\CCR\Documents\', FN));
  WriteLn(ExtractRelativePath('C:\Users\CCR\Downloads\', FN));
  WriteLn(ExtractRelativePath('C:\Users\CCR', FN));
```

Given the caveat just mentioned, the above code outputs the following:

```
Documents\File.txt
File.txt
..\Documents\File.txt
CCR\Documents\File.txt
```

To go the other way, there are three functions to choose from: `ExpandFileName`, `ExpandUNCFileName` and `ExpandFileNameCase`. The last of these is irrelevant on Windows, given file names there are case insensitive; however, it is still available to use. As for `ExpandFileName` and `ExpandUNCFileName`, they only differ in a network situation: if a certain network resource has been mapped to a Windows drive letter, `ExpandFileName` *won't* convert from drive letter (e.g. `H:`) to the network share address (`\\servername\share`) where `ExpandUNCFileName` *will*.

Annoyingly enough, while there are three functions to go from a relative to an absolute path, none of them allow you to specify the base path explicitly — instead, the current working directory is used. Given the latter in a GUI application can be easily changed, you'll probably have to always call `SetCurrentDir` or `TDirectory.SetCurrentDirectory` first:

```
function ExpandFileName(const BasePath,
  FileName: string): string; overload;
var
  SavedDir: string;
begin
  SavedDir := TDirectory.GetCurrentDirectory;
  TDirectory.SetCurrentDirectory(BasePath);
  Result := ExpandFileName(FileName);
  TDirectory.SetCurrentDirectory(SavedDir);
end;
```

Aside from the functions already mentioned, the `TPath` record also provides a simple `Combine` method, which prepends one path to another so long as the second is not rooted (if it is, then it is just returned unchanged):

```
{ The next line returns C:\Base\Sub\File.txt }
S := TPath.Combine('C:\Base', 'Sub\File.txt');
{ The next line returns D:\File.txt }
S := TPath.Combine('C:\Base', 'D:\File.txt');
{ The next line returns C:\Base\..\File.txt }
S := TPath.Combine('C:\Base', '..\File.txt');
```

Notice `TPath.Combine` doesn't have any special handling of the 'go up a level' specifier (`..\`), unlike `ExpandFileName` and friends.

8. Advanced I/O: streams

A ‘stream’ is an object for reading or writing data, doing so in a manner that abstracts away from the underlying medium, be that a file on disk, dynamically allocated memory block, database BLOB field, or so on. In the Delphi RTL, this gives rise to an abstract base class (`TStream`) and several descendant classes (`TFileStream`, `TMemoryStream`, `TBlobStream`, and so on): where `TStream` defines the interface for you to work against, the descendants map it onto particular media.

Notwithstanding their inherent abstraction, streams are still relatively low-level objects. Rather than asking a stream to read or write an XML structure, for example, you ask it to read or write a specified number of bytes. Consequently, this chapter covers more advanced ground than either its predecessor, where we looked at manipulating files at the file system level, or its sequel, which will turn to working with things like XML *as* XML.

In the present chapter, in contrast, we will investigate streams in depth: how to use them, the different sorts of stream and stream helper classes available ‘in the box’, and the ways in which you can extend and deepen stock stream functionality with some custom code.

Stream nuts and bolts

The core members of TStream

When used directly, streams are things you can read from, write to, and 'seek' across. To that effect, `TStream`'s core public members are the following:

```
type
  TSeekOrigin = (soBeginning, soCurrent, soEnd);

function Read(var Buffer; Count: Longint): Longint; virtual;
procedure ReadBuffer(var Buffer; Count: Longint);
function Write(const Buffer; Count: Longint): Longint; virtual;
procedure WriteBuffer(const Buffer; Count: Longint);
function Seek(const Offset: Int64;
  Origin: TSeekOrigin): Int64; virtual;
function CopyFrom(Source: TStream; Count: Int64): Int64;
property Position: Int64 read GetPosition write SetPosition;
property Size: Int64 read GetSize write SetSize64;
```

`Read` aims to copy the specified number of bytes into a user-provided buffer, returning the actual number of bytes copied. This may be less than the requested amount depending on how much data was left in the stream to read. `ReadBuffer` does the same as `Read`, only instead of returning the number of bytes read, raises an `EReadError` exception if the requested number of bytes couldn't be returned.

`Write` and `WriteBuffer` have the same semantics of `Read` and `ReadBuffer`, only in the context of writing rather than reading. Since in all cases the `Buffer` parameter is untyped, there are no restrictions on what the compiler allows you to pass for it. If you pass a string or dynamic array, you should pass the first element, not the string or array directly. Alternatively, you can cast to the corresponding pointer type and dereference; while the resulting code is a bit uglier, it removes the need to check the string or array isn't empty:

```
procedure StreamBytes(Stream: TStream; const Bytes: TBytes);
begin
  if Length(Bytes) <> 0 then //or 'if Bytes <> nil then'
    Stream.Write(Bytes[0], Length(Bytes));
end;

procedure StreamBytes2(Stream: TStream; const Bytes: TBytes);
begin
  Stream.Write(PByte(Bytes)^, Length(Bytes));
end;

procedure StreamUTF8Str(Stream: TStream; const S: UTF8String);
begin
  if Length(S) <> 0 then Stream.Write(S[1], Length(S));
end;

procedure StreamUTF8Str2(Stream: TStream; const S: UTF8String);
begin
  Stream.Write(PAnsiChar(S)^, Length(S));
end;
```

Similarly, if you pass a pointer of some kind, you must dereference it:

```
var
  Buffer: Pointer; //or PAnsiChar, PWideChar, PByte, etc.
begin
  GetMem(Buffer, BufferSize);
  try
    while MyStream.Read(Buffer^, BufferSize) > 0 do
    begin
      //work with Buffer
    end;
```

Failure to dereference is not a compiler error since a pointer is still a 'value' of some kind, and therefore, acceptable as an untyped parameter. Nonetheless, what should be streamed in or out is the data pointed to, not the pointer itself.

Another thing to keep in mind is that the `Count` parameter of `Read` and friends is in terms of *bytes*. This may imply doing some simple arithmetic to convert from and to the number of *elements*:

```
procedure StreamDoubles(Stream: TStream; const Data: array of Double);
begin
  Stream.Write(Data[0], Length(Data) * SizeOf(Double));
end;
```

Of the other TStream methods, Seek moves the 'seek pointer', meaning the position in the stream at which data is read from or written to, without reading any data itself. For example, to move the seek pointer on by 40 bytes, use Stream.Seek(40, soCurrent), and to move it to 20 bytes before the stream's end, Stream.Seek(-20, soEnd). Normally, the value for Offset when Origin is soEnd would be either 0 or a negative figure. However, depending on the stream source, it can be possible to seek to somewhere beyond the end of a stream.

To retrieve the location of the seek pointer relative to the start of the stream, use the Position property. Position is also writeable, in which case it does the same thing as calling Seek with soBeginning as the second parameter.

The Size property returns (and sometimes is able to set) the total size of the stream. Examples of TStream descendants that allow you to set their Size property include TMemoryStream and TFileStream. Here, setting Size to something smaller than its current value will truncate, and setting it to something larger will leave the new portion of the stream with unspecified values.

Lastly, CopyFrom copies the specified number of bytes from one stream into another. The Count argument can be as large as you like: internally, CopyFrom will allocate a temporary memory buffer to act an intermediary, repeatedly refilling it as necessary. If Count is 0 however, the source stream has its Position property set to 0 before its complete contents are copied over. Watch out for this, since if the value you pass for Count is the result of some calculation, it is easy to pass a zero value on the wrong assumption 0 means 'don't copy anything'.

TStream in practice

The most basic use of TStream doesn't involve calling any of its methods directly. Instead, you call another object's LoadFromStream or SaveToStream method, passing a TStream instance to it. These methods are defined by classes such as TBitmap, TStrings and TXMLDocument:

```
uses System.Types;

procedure TfrmMain.btnTestClick(Sender: TObject);
var
  Stream: TStream;
begin
  //load a memo's contents from a resource stream
  Stream := TResourceStream.Create(HInstance, 'HELP', RT_RCDATA);
  try
    memHelp.Lines.LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
  //append a memo's contents to a file
  if FileExists('C:\Foo.txt') then
    Stream := TFileStream.Create('C:\Foo.txt', fmOpenReadWrite)
  else
    Stream := TFileStream.Create('C:\Foo.txt', fmCreate);
  try
    Stream.Seek(0, soEnd);
    memOutput.Lines.SaveToStream(Stream);
  finally
    Stream.Free;
  end;
end;
```

Internally, LoadFromStream and SaveToStream will call the stream's Read/ReadBuffer, Write/WriteBuffer and Seek methods as appropriate. Let us now look at a concrete example of doing that ourselves.

Reading and patching an EXE or DLL's timestamp

Imagine wanting to learn when exactly a Windows EXE or DLL was compiled. This can be done by parsing the file's header structures. While this method is not foolproof (one of the reasons being the timestamp can be easily patched, as we shall be seeing!), it usually gives an accurate result given the most popular compilers for Windows output it correctly.

Formally, Win32 and Win64 EXEs and DLLs adhere to Microsoft's 'PE' (Portable Executable) standard. You can find documentation for this file format on MSDN (in particular, look for articles by Matt Pietrek), though for our purposes we just need to understand the dummy DOS header and the initial 'NT' one: the DOS header links to the NT one, which then contains the timestamp in Unix format (the PE format is a development of an older Unix standard, though confusingly enough, not one modern *nix systems like Linux and OS X themselves use). In Delphi, record types and constants for these structures are declared in the Winapi.Windows unit. Helpfully, System.SysUtils also provides a function (UnixToDateTime) to convert between a Unix date/time and a Delphi TDateTime. Putting these things together, a function for

returning an EXE or DLL's timestamp can be written like this:

```
uses Winapi.Windows, System.SysUtils, System.Classes;

function PEHeaderDateTimeUTC(const FileName: string): TDateTime;
var
  DOSHeader: TImageDosHeader;
  NTHeader: TImageNtHeaders;
  Stream: TFileStream;
begin
  Result := 0;
  //open a file stream in read mode
  Stream := TFileStream.Create(FileName,
    fmOpenRead or fmShareDenyWrite);
  try
    //read the DOS header and validate it
    Stream.ReadBuffer(DOSHeader, SizeOf(DOSHeader));
    if DOSHeader.e_magic <> IMAGE_DOS_SIGNATURE then
      raise EParserError.Create('Invalid PE file');
    //seek to the NT header, read and validate it
    Stream.Seek(DOSHeader._lfanew, soBeginning);
    Stream.ReadBuffer(NTHeader, SizeOf(NTHeader));
    if NTHeader.Signature <> IMAGE_NT_SIGNATURE then
      raise EParserError.Create('Invalid PE file');
    //read off the timestamp and convert it to a TDateTime
    Result := UnixToDateTime(NTHeader.FileHeader.TimeDateStamp);
  finally
    Stream.Free;
  end;
end;
```

As the timestamp is stored using the UTC/GMT time zone, it will need to be converted to local time for display purposes:

```
uses System.DateUtils;

function PEHeaderDateTime(const FileName: string): TDateTime;
begin
  Result := TTimeZone.Local.ToLocalTime(PEHeaderDateTimeUTC(FileName));
end;

var
  DT: TDateTime;
begin
  DT := PEHeaderDateTime(GetModuleName(0));
  WriteLn('This EXE file was compiled on ',
    FormatDateTime('dddddd "at" t', DT));
```

Patching the timestamp can be done similarly:

```
procedure PatchPEHeaderDateTime(const FileName: string;
  NewValue: TDateTime; IsLocal: Boolean = True);
var
  DOSHeader: TImageDosHeader;
  NTHeader: TImageNtHeaders;
  Stream: TFileStream;
begin
  //convert the new value to UTC/GMT if necessary
  if IsLocal then
    NewValue := TTimeZone.Local.ToUniversalTime(NewValue);
  //open a file stream in read/write mode
  Stream := TFileStream.Create(FileName, fmOpenReadWrite);
  try
    //read the DOS header and validate it
    Stream.ReadBuffer(DOSHeader, SizeOf(DOSHeader));
    if DOSHeader.e_magic <> IMAGE_DOS_SIGNATURE then
      raise EParserError.Create('Invalid PE file');
    //seek to the NT header, read and validate it
    Stream.Seek(DOSHeader._lfanew, soBeginning);
    Stream.ReadBuffer(NTHeader, SizeOf(NTHeader));
    if NTHeader.Signature <> IMAGE_NT_SIGNATURE then
      raise EParserError.Create('Invalid PE file');
    //update the structure just read in
    NTHeader.FileHeader.TimeDateStamp := DateTimeToUnix(NewValue);
    //seek back on ourselves
    Stream.Seek(-SizeOf(NTHeader), soCurrent);
    //write the revised record structure
    Stream.WriteBuffer(NTHeader, SizeOf(NTHeader));
  finally
```

```
    Stream.Free;
  end;
end;
```

If this code looks intimidating, try to break it down:

- Since in the patching case we are editing the contents of an existing file, we need to create a file stream in read/write mode.
- The first few bytes of the file should be structured according to the `TImageDosHeader` record, so we read that part of the file in.
- At a certain position in the `TImageDosHeader` structure, a value denoted by the `IMAGE_DOS_SIGNATURE` constant will appear if the file is a valid PE file; if it isn't found, raise an exception.
- Seek to the position of the 'NT header' structure, and read it in.
- Check the structure's 'signature' is present and correct.
- Write the new timestamp to the NT header record.
- Seek back to where the NT header began, and overwrite the existing header with the revised one.
- Free the stream.

Streaming issues

Ideally, streaming any sort of variable would be like streaming the `TImageDosHeader` and `TImageNtHeaders` records in the previous example, which were passed directly to `ReadBuffer` and `WriteBuffer`. Unfortunately, only a small subset of types are directly streamable in that way though: specifically, only simple value types, e.g. numbers and enumerations, together with static arrays and records whose elements or fields are themselves only value types, are. (`TImageDosHeader` and `TImageNtHeaders` are OK since they fall into the last group.)

Another potential problem to consider is the size of the types streamed. In the case of numbers, it is prudent to make this explicit by streaming types with a fixed size: `Int32` not `Integer`, `UInt32` not `Cardinal`, `Double` not `Real`, etc. The reason for this is that the size of the generic type may change over time or platform, so in any case where you are streaming data to and from a file (for example), using fixed size types will ensure the same code can read and write files produced on different platforms and at different times.

A related issue concerns streaming records. Consider the following code:

```
type
  TTestRec = record
    First: Int8;
    Second: Int16;
  end;

begin
  WriteLn(SizeOf(TTestRec));
end.
```

If you think the output will be 3 given an `Int8` is one byte long and an `Int16` two, you would be wrong: instead, size of the record is 4. This is because the compiler inserts a hidden, one byte field before `Second` to ensure the record is 'aligned'. Modern processors have an increasing preference for aligned data structures, so if you can, just go with the flow. However, you can also disable record alignment via the `packed` keyword:

```
type
  TTestRec = packed record
    First: Int8;
    Second: Int16;
  end;
```

Slightly greater control is available with the `$A` compiler directive: `{$A1}` disables alignment completely, which is the same as using `packed` on a specific type, `{$A2}` aligns to two byte boundaries, `{$A4}` to four and `{$A8}` to eight.

Streaming strings and dynamic arrays

We have already seen how extra care needs to be taken when streaming strings and dynamic arrays — that is, you must pass their first element rather than the string or array directly, and you must be careful to specify the number of bytes rather than the number of elements. In practice, another thing to consider is how the reading code is supposed to know how many elements to read, given the length of a string or dynamic array, by their very nature, is not determined

'statically' at compile time. An easy solution to this problem is to write the length out immediately before the data:

```
procedure SaveDoublesToStream(const Values: TArray<Double>;
  Stream: TStream);
var
  Len: Int32;
begin
  Len := Length(Values);
  Stream.WriteBuffer(Len, SizeOf(Len));
  if Len > 0 then
    Stream.WriteBuffer(Values[0], Len * SizeOf(Values[0]));
end;

function LoadDoublesFromStream(Stream: TStream): TArray<Double>;
var
  Len: Int32;
begin
  Stream.ReadBuffer(Len, SizeOf(Len));
  SetLength(Result, Len);
  //NB: you may wish to raise an exception if Len < 0
  if Len > 0 then
    Stream.ReadBuffer(Result[0], Len * SizeOf(Result[0]));
end;
```

Since `Double` (like `Char`) is a directly streamable type, we can stream out the list of values in one go. Streaming an array of another reference type is more tedious though; in that case, elements must be written and read individually:

```
procedure SaveStringArrayToStream(const Values: array of string;
  Stream: TStream);
var
  Len: Int32;
  S: string;
begin
  Len := Length(Values);
  Stream.WriteBuffer(Len, SizeOf(Len));
  if Len = 0 then Exit;
  for S in Values do
  begin
    Len := Length(S);
    Stream.WriteBuffer(Len, SizeOf(Len));
    if Len <> 0 then
      Stream.WriteBuffer(S[1], Len * SizeOf(S[1]));
  end;
end;

function LoadStringArrayFromStream(Stream: TStream): TArray<string>;
var
  I, Len: Int32;
begin
  Stream.ReadBuffer(Len, SizeOf(Len));
  SetLength(Result, Len);
  //NB: you may wish to raise an exception if Len < 0
  if Len <= 0 then Exit;
  for I := 0 to Len - 1 do
  begin
    Stream.ReadBuffer(Len, SizeOf(Len));
    //again, an exception might be raised if Len < 0
    if Len > 0 then
      Stream.ReadBuffer(Result[I][1], Len * SizeOf(Char));
  end;
end;
```

In principle, the same item-by-item approach is necessary when streaming classes or records containing managed types too. However, components specifically have dedicated support built into `TStream` itself, functionality that we will be looking at later in this chapter.

With respect to strings again, another aspect to consider is the format they are streamed out in. In the previous example, the streaming format was just the in-memory format, which means UTF-16 was used. A much more popular encoding in practice is UTF-8 though, due to the fact it is usually more space efficient than UTF-16. In order to stream to and from UTF-8 instead of UTF-16, you need to use a `UTF8String` variable as an intermediary:

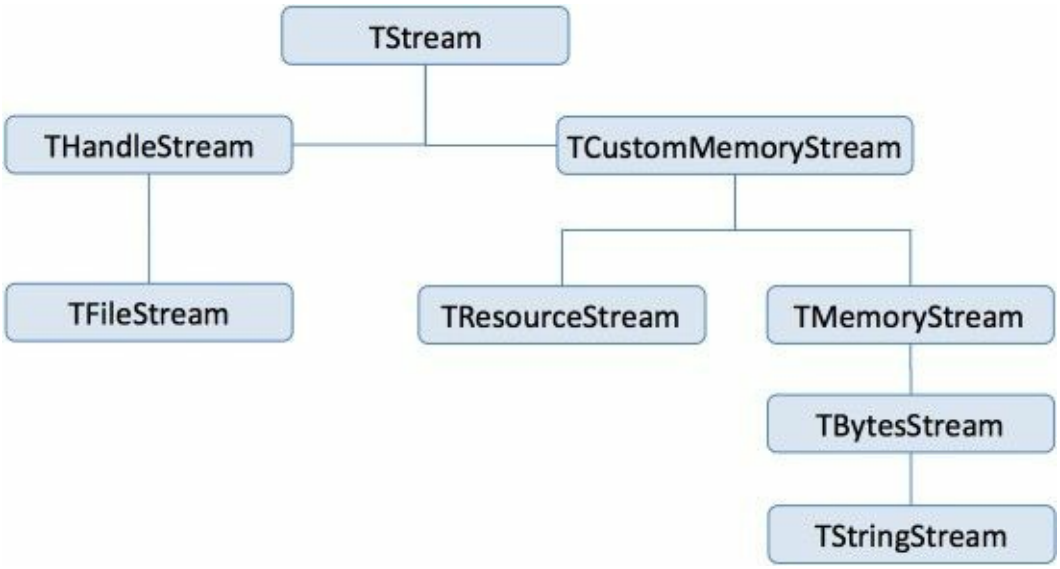
```
procedure WriteStringAsUTF8(const S: string; Stream: TStream);
var
  Len: Int32;
  UTF8: UTF8String;
begin
```

```
  UTF8 := UTF8String(S);
  Len := Length(UTF8); //*not* Length(S)!
  Stream.WriteBuffer(Len, SizeOf(Len));
  if Len > 0 then Stream.WriteBuffer(UTF8[1], Len);
end;

function ReadStringAsUTF8(Stream: TStream): string;
var
  Len: Int32;
  UTF8: UTF8String;
begin
  Stream.ReadBuffer(Len, SizeOf(Len));
  if Len <= 0 then Exit('');
  SetLength(UTF8, Len);
  Stream.ReadBuffer(UTF8[1], Len);
  Result := string(UTF8);
end;
```

Multiplying the length by the size of an element isn't required in the `UTF8String` case, since each element of this type is one byte in size.

Types of stream

`TStream` is not a class you create directly. Rather, you create one of its descendants. The `System.Classes` unit implements seven (more are defined in other units):



We'll look at each of these in turn.

Handle streams

On Windows, `THandleStream` wraps the `ReadFile`/`WriteFile` API; on the Mac, the `__read`/`__write` POSIX functions. When instantiating it, you pass the constructor a 'handle' got from an API function, e.g. `CreateFile` or `GetStdHandle` if targeting Windows, and `__open` if targeting OS X. Whatever handle you pass can be retrieved later via the `Handle` property of the constructed object:

```
uses
  System.Classes, Winapi.Windows;

var
  PipeHandle: THandle;
  PipeStream: THandleStream;
begin
  PipeHandle := CreateNamedPipe('\\.\pipe\MyCoolApp',
    PIPE_ACCESS_INBOUND, 0, PIPE_UNLIMITED_INSTANCES, 1024,
    1024, 0, nil);
  PipeStream := THandleStream.Create(PipeHandle);
```

The stream doesn't take ownership of the handle. This means that if the handle requires releasing when finished with (normally it will do), you must call the relevant API function yourself immediately before freeing the stream:

```
CloseHandle(PipeStream.Handle);
PipeStream.Free;
```

File streams

`TFileStream` inherits from `THandleStream`, however it abstracts from the native API completely by taking a file name and a Delphi-defined numeric 'mode' in its constructor. This 'mode' comes in two parts, one required and the other optional; if both are specified, combine them using the `or` operator.

The first element can be one of `fmCreate`, `fmOpenRead`, `fmOpenWrite` or `fmOpenReadWrite`:

- Pass `fmCreate` to create a new file, wiping any data if one already exists.
- Pass `fmOpenRead` if you need just read-only access to an existing file.
- Pass `fmOpenWrite` if you need just write-only access to an existing file.
- Pass `fmOpenReadWrite` if you need read and write access to an existing file.

When passing `fmCreate`, combine it with `fmExclusive` if you wish an exception to be raised if the file already exists. This is useful when creating a lock file of some sort. For `fmOpenRead`, `fmOpenWrite` and `fmOpenReadWrite`, the options are one of `fmShareExclusive`, `fmShareDenyWrite` or `fmShareDenyNone` — on Windows, there is also `fmShareDenyRead`. The usual case for the write modes is not to specify anything else, however when using `fmOpenRead`, it is a good idea to combine it with

`fmShareDenyWrite`. This enables other applications to open (but not modify) the file while you yourself have it open:

```
var
  Stream: TFileStream;
begin
  Stream := TFileStream.Create('MyFile.txt', fmOpenRead or fmShareDenyWrite);
  try
    //read stuff...
  finally
    Stream.Free;
  end;
end;
```

If you don't specify a sharing mode at all, then the default, most restrictive mode is applied. This can be rather user-hostile, particularly when opening the file wasn't at the user's instigation in the first place.

Memory streams

Next to `TFileStream`, the most commonly-used stream class is probably `TMemoryStream`. This puts the `TStream` interface over an in-memory buffer, managed by the stream itself. In addition to the ordinary stream members, it and its immediate ancestor, `TCustomMemoryStream`, add the following:

```
{ The first three are introduced by TMemoryStream }
procedure Clear;
procedure LoadFromStream(Stream: TStream);
procedure LoadFromFile(const FileName: string);
{ The last three are introduced by TCustomMemoryStream }
procedure SaveToStream(Stream: TStream);
procedure SaveToFile(const FileName: string);
property Memory: Pointer read FMemory;
```

Both `LoadFromStream` and `LoadFromFile` clear any existing data before copying the contents of the source stream or file. Conversely, `SaveToStream` and `SaveToFile` copy the complete contents of the memory stream to the destination. If you want to copy just the data from the current seek pointer position onwards instead, a little extra coding is needed:

```
procedure SaveRestOfMemStream(Source: TCustomMemoryStream;
  Dest: TStream); overload;
var
  Ptr: PByte;
begin
  Ptr := PByte(Source.Memory);
  Inc(Ptr, Source.Position);
  Dest.WriteBuffer(Ptr^, Source.Size - Source.Position);
end;

procedure SaveRestOfMemStream(Source: TCustomMemoryStream;
  const Dest: string); overload;
var
  DestStream: TFileStream;
begin
  DestStream := TFileStream.Create(Dest, fmCreate);
  try
    SaveRemainderOfMemoryStream(Source, DestStream);
  finally
    DestStream.Free;
  end;
end;
```

The actual amount of memory used by a `TMemoryStream` is independent of what the `Size` property reports — internally, it requests memory in 8KB blocks. This is for efficiency and performance reasons: if you write a large amount of data over many `Write` or `WriteBuffer` calls, it is better to allocate a destination buffer larger than is needed up front than to continually reallocate as you go along.

The amount of memory a `TMemoryStream` instance has allocated for itself can be read and changed via the `Capacity` property. Unfortunately, this property has only `protected` visibility however, meaning you must either subclass `TMemoryStream` or use an interposer class to access it:

```
type //define interposer class
  TMemoryStreamAccess = class(TMemoryStream);

procedure ChangeMemStreamCapacity(Stream: TMemoryStream;
  NewCapacity: Int32);
begin
  TMemoryStreamAccess(Stream).Capacity := NewCapacity;
end;
```

Changing `Capacity` to use a different policy than the default may not actually speed up your program however. This is because `TMemoryStream` requests memory from an application or system-wide memory manager that will itself have an allocation strategy that second-guesses future requests.

TBytesStream and TStringStream

`TBytesStream` is a simple descendent of `TMemoryStream` that uses (and exposes) a `TBytes` (= dynamic array of `Byte`) backing store. This is mainly useful if you already have a `TBytes` instance that you wish to use through the `TStream` interface:

```
procedure LoadBitmapFromBytes(Bitmap: TBitmap; const Bytes: TBytes);
var
  Stream: TBytesStream;
begin
  Stream := TBytesStream.Create(Bytes);
  try
    Bitmap.LoadFromStream(Bytes);
  finally
    Stream.Free;
  end;
end;
```

It is OK to pass `nil` to the `TBytesStream` constructor; if you do, a new `TBytes` instance will be created on `Write`, `WriteBuffer` or `CopyFrom` being called.

`TStringStream` descends from `TBytesStream` to add methods exposing the raw bytes as strings with a certain encoding:

```
{ constructors }
constructor Create; overload;
constructor Create(const AString: string); overload;
constructor Create(const AString: RawByteString); overload;
constructor Create(const AString: string; AEncoding: TEncoding;
  AOwnsEncoding: Boolean = True); overload;
constructor Create(const AString: string;
  ACodePage: Integer); overload; //pass, for example, CP_UTF8
constructor Create(const ABytes: TBytes); overload;

{ methods and properties }
function ReadString(Count: Longint): string;
procedure WriteString(const AString: string);
property DataString: string read GetDataString;
property Encoding: TEncoding read FEncoding;
```

The encoding determines how character data is both read from the `TBytes` backing store and written to it (note this is literally just character data — any length markers must still be written out manually). If you do not specify an encoding in the constructor, be this explicitly with a `TEncoding` instance or implicitly with the `RawByteString` overload, then `TEncoding.Default` will be used. This behaviour is highly likely to result in a different encoding being employed between Windows and OS X, and even different Windows installations — on OS X, UTF-8 will be used, but on Windows, whatever is the current system default legacy 'Ansi' code page will be. Because of this, you should generally always explicitly specify a `TEncoding` instance in the constructor.

E.g., to create an empty `TStringStream` instance whose `WriteString` method writes UTF-8, you should do this:

```
Stream := TStringStream.Create('', TEncoding.UTF8);
```

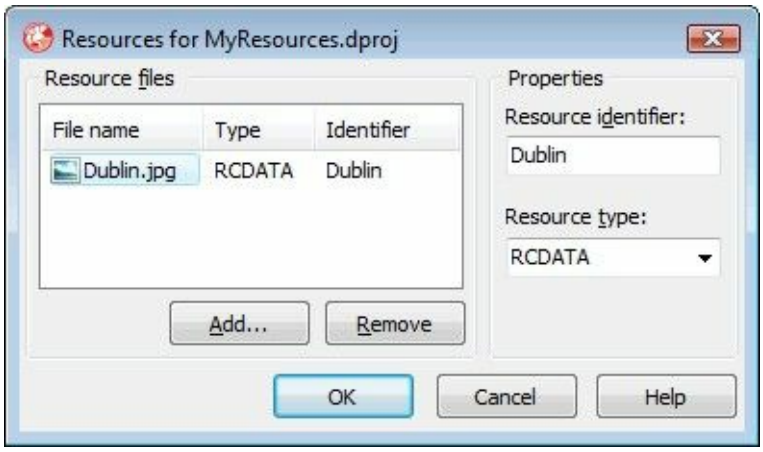
Alternatively, you might just avoid `TStringStream` in the first place and use code that employs a `UTF8String` as an intermediary instead, as was demonstrated earlier.

Resource streams

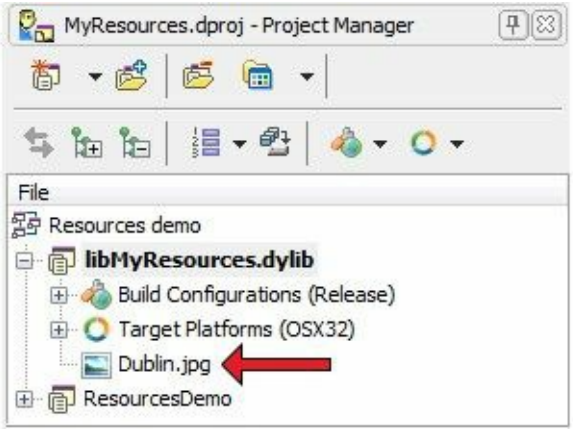
`TResourceStream` provides read-only access to embedded resources. The sort of 'resource' meant here is a file (typically a picture, but could be anything) that has been embedded inside a program or library in a special way.

On Windows, the way resources are embedded is defined by the operating system. This leads to `TResourceStream` delegating to the operating system's resource API to do its grunt work. On OS X, in contrast, there is no such API, since the convention there to leave resources as standalone files, albeit still within the application 'bundle'. Nonetheless, when targeting the Mac, the Delphi compiler and RTL provide Windows-like embedding functionality, allowing you to use resources on the Mac in exactly the same way as you would on Windows.

To embed files in the first place, select `Project|Resources and Images...` from the IDE's menu bar. This will lead a simple dialog being shown, from which you can add the files to embed:



Each file must be given a textual identifier and a type; usually, the type is left as `RCDATA`, unless you are targeting only Windows, in which case `ICON` should be used for a Windows *.ico file, `BITMAP` for a *.bmp, and `CURSOR` for a *.cur (the `FONT` option is for *.fon files, which are pretty rare nowadays). After OK'ing the dialog, the files just added will become listed in the Project Manager to the right of the screen:



Don't subsequently delete the source files, since only links are stored by the project until compile time.

Loading resources at runtime

To load a resource at runtime, pass the 'module handle', resource identifier and resource type to the `TResourceStream` constructor. For example, say you added a PNG image to the project, called it `MyPng`, and gave it a resource type of `RCDATA`. The following code will now load it into a FireMonkey `TImage` control:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Stream: TResourceStream;
begin
  Stream := TResourceStream.Create(HInstance, 'MyPng', RT_RCDATA);
  try
    Image1.Bitmap.LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
end;
```

As shown, a resource type of `RCDATA` becomes `RT_RCDATA` in code. In the case of a VCL application, a resource type of `BITMAP` will become `RT_BITMAP` and so on.

If a resource cannot be found, then an `EResNotFound` exception will be raised. To allow for the resource not being there, call the `FindResource` function before creating the stream. On Windows this routine is defined in the `Winapi.Windows` unit, on other platforms the `System` unit. If the function returns 0, then the resource is missing; any other value, and it exists:

```
procedure TForm1.LoadResButtonClick(Sender: TObject);
var
  Stream: TResourceStream;
begin
  if FindResource(HInstance, 'MyPng', RT_RCDATA) <> 0 then
  begin
    Stream := TResourceStream.Create(HInstance,
      'MyPng', RT_RCDATA);
    try
```

```
      Image1.Bitmap.LoadFromStream(Stream);
    finally
      Stream.Free;
    end;
  end;
end;
```

Resource libraries

When you call `TResourceStream.Create`, passing `HInstance` for the first argument instructs the class to look in the current module for the resource, which in the example just given means the executable. In principle, resources could be embedded inside dynamic libraries or packages too however (DLLs/BPLs on Windows, dylibs on the Mac). Because of this, Windows has the concept of 'resource-only DLLs', which instead of exporting procedures or functions, contain just resources instead.

To see this in action, create a new dynamic link library project in the IDE (`File|New|Other...`, `Delphi Projects` node, `Dynamic-link Library`). In the DPR, delete the default comment and uses clause to leave it looking like this:

```
library Project1;

{$R *.res}

begin
end.
```

Next, add a JPEG image of your choice to the project's resources list via `Project|Resources and Images...`; name the resource `MyJPEG`, and leave the resource type as `RCDATA`.

Now add a FireMonkey HD application project, either via `Project|Add Add New Project...`, or by right clicking on the `ProjectGroup1` node at the top of the Project Manager and selecting `Add New Project...` from the popup menu. Once created, add a `TImage` and `TButton` to the form, size the image as appropriate, and handle the button's `OnClick` event like this:

```
{$IFDEF MSWINDOWS}
uses Winapi.Windows;
{$ENDIF}

const
  LibName = {$IFDEF MSWINDOWS}'MyResources.dll'
            {$ELSE}'libMyResources.dylib'
            {$ENDIF};

procedure TForm1.Button1Click(Sender: TObject);
var
  Lib: HMODULE;
  Stream: TResourceStream;
begin
  Stream := nil;
  Lib := LoadLibrary(LibName);
  if Lib = 0 then
    raise EFileNotFoundException.Create('Library not found');
  try
    Stream := TResourceStream.Create(Lib, 'MyJPEG', RT_RCDATA);
    Image1.Bitmap.LoadFromStream(Stream);
  finally
    Stream.Free;
    FreeLibrary(Lib);
  end;
end;
```

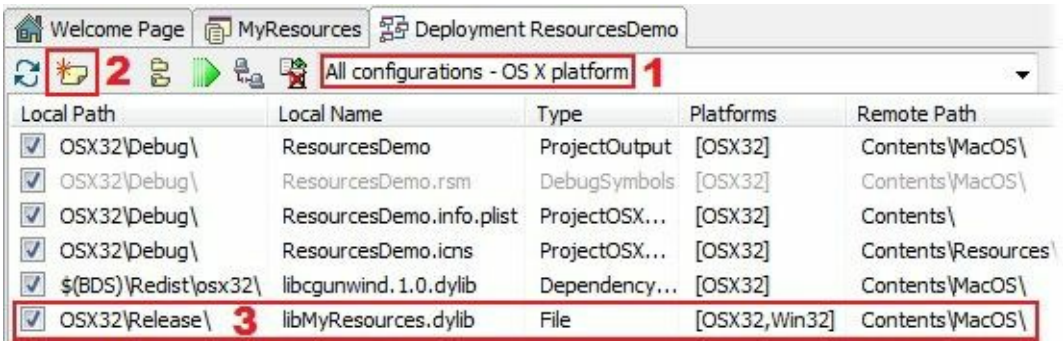
Finally, save everything to the same directory: the library project as `MyResources.dpr`, the program project as `ResourcesDemo.dpr`, and the project group as you like. Using the default Win32 target, compile the library, then run the program; the button should load the image from the DLL as expected.

The crucial point in the code just presented is that the library is freed only *after* the resource stream is finished with. In principle, you could load a resource library at startup, delay loading a whole range of resources until a later point, then free the library at shutdown. What you must not do, however, is load a library, create a resource stream on it, then immediately free the library before going on to read from the stream. If you want to free the library straight away, copy the resource stream data to a memory stream first:

```
function CreateResourceStreamForLater(Inst: HINST;
  const ResName: string): TMemoryStream;
var
```

```
  ResStream: TResourceStream;
begin
  ResStream := TResourceStream.Create(Inst, ResName);
  try
    Result := TMemoryStream.Create;
    Result.CopyFrom(ResStream, 0);
  finally
    ResStream.Free;
  end;
end;
```

When targeting the Mac, you will need to do one extra thing compared to Windows, which is to add the compiled library (dylib) as an explicit dependency for the executable to deploy. Do this by adding the path to the dylib to the executable project's deployment list, as displayed when you choose `Project|Deployment` from the main menu bar in the IDE:



Depending on the library's active build configuration, the dylib will be outputted to either `%LibProjectDir%\OSX\Debug` or `%LibProjectDir%\OSX\Release`. I suggest you always compile it using the Release configuration, since Debug will bulk things up for zero benefit (it is perfectly OK to compile the library using one build configuration and the executable another).

Reader and writer classes

'Readers' and 'writers' are classes that overlay a stream to make reading and writing specific types easier. Three pairs are declared in `System.Classes`: `TBinaryReader`/`TBinaryWriter`, for simple types; `TStreamReader`/`TStreamWriter`, for reading and writing streams solely composed of textual information; and `TReader`/`TWriter`, for reading and writing data in a fashion that stores basic type information too.

TBinaryReader/TBinaryWriter

`TBinaryReader` and `TBinaryWriter` provide a thin layer on top of `TStream` for the purpose of directly reading and writing variables typed to `Boolean`, `string`, and the various integer types. In use, you pass their constructors either an existing stream to work on or a file name, in which case a file stream will be constructed internally, then call `ReadXXX` (`TBinaryReader`) or `Write` (`TBinaryWriter`) as appropriate. For example, imagine you have an array of the following record type you want to stream:

```
uses
  System.SysUtils, System.Classes;

type
  TPerson = record
    Forename, Surname: string;
    DateOfBirth: TDate; //NB: TDate = type Double
  end;
```

To stream out, you might write a helper routine like this:

```
procedure WritePeopleToStream(const People: TArray<TPerson>;
  Dest: TStream);
var
  I, Len: Int32;
  Writer: TBinaryWriter;
begin
  Writer := TBinaryWriter.Create(Dest);
  try
    Len := Length(People);
    Writer.Write(Len);
    for I := 0 to Len - 1 do
    begin
      Writer.Write(People[I].Forename);
      Writer.Write(People[I].Surname);
      Writer.Write(People[I].DateOfBirth);
    end;
  finally
    Writer.Free;
  end;
end;
```

And to stream back in, a function like this:

```
function ReadPeopleFromStream(Source: TStream): TArray<TPerson>;
var
  I: Int32;
  Reader: TBinaryReader;
begin
  Reader := TBinaryReader.Create(Source);
  try
    SetLength(Result, Reader.ReadInt32);
    for I := 0 to High(Result) do
    begin
      Result[I].Forename := Reader.ReadString;
      Result[I].Surname := Reader.ReadString;
      Result[I].DateOfBirth := Reader.ReadDouble;
    end;
  finally
    Reader.Free;
  end;
end;
```

Constructing a TBinaryReader or TBinaryWriter

As declared, `TBinaryReader`'s constructors look like this:

```
constructor Create(Stream: TStream; AEncoding: TEncoding = nil);
constructor Create(Stream: TStream; AEncoding: TEncoding;
  AOwnsStream: Boolean = False);
```

```
constructor Create(const FileName: string; Encoding: TEncoding = nil);
```

If no encoding is specified, then `TEncoding.UTF8` serves as the default. Pass `True` for `AOwnsStream` if you want the reader to free the stream when the reader itself is destroyed.

The constructors for `TBinaryWriter` correspond to `TBinaryReader`'s, only with an additional parameter in the file name case to allow specifying whether the file should be appended to or simply replaced:

```
constructor Create(Stream: TStream);
constructor Create(Stream: TStream; Encoding: TEncoding);
constructor Create(Stream: TStream; Encoding: TEncoding;
  AOwnsStream: Boolean);
constructor Create(const Filename: string;
  Append: Boolean = False);
constructor Create(const Filename: string; Append: Boolean;
  Encoding: TEncoding);
```

Both `TBinaryReader` and `TBinaryWriter` provide a `BaseStream` property to get to the underlying stream:

```
if Reader.BaseStream is TFileStream then
  Write('Writing to ', TFileStream(Reader.BaseStream).FileName);
```

Reading and writing values

To read a value using a `TBinaryReader`, call one of `ReadBoolean`, `ReadByte`, `ReadChar`, `ReadDouble`, `ReadShortInt`, `ReadInteger`, `ReadInt32`, `ReadInt64`, `ReadSmallInt`, `ReadSingle`, `ReadString`, `ReadWord`, `ReadUInt16`, `ReadCardinal`, `ReadUInt32` and `ReadUInt64`. All but `ReadChar` and `ReadString` are implemented as trivial calls to the underlying stream's `ReadBuffer` method — no length markers are outputted, for example.

Slightly more complicated are `PeekChar`, `Read`, `ReadChar`, `ReadChars` and `ReadString`:

```
function PeekChar: Integer; virtual;
function Read: Integer; overload; virtual;
function Read(const Buffer: TCharArray;
  Index, Count: Integer): Integer; overload; virtual;
function ReadChar: Char; virtual;
function ReadChars(Count: Integer): TCharArray; virtual;
function ReadString: string; virtual;
```

All make use of the `TEncoding` instance passed to the constructor:

- The parameterless version of `Read` reads the next character as defined by that encoding and returns the character's Unicode ordinal value (e.g., 97 for 'a', and 981 for the Greek symbol 'ϕ'). If the stream's seek pointer was already at the stream's end, then -1 is returned. *[NB: due to a bug, this method is broken if TEncoding.Unicode was not passed to the constructor (http://qc.embarcadero.com/wc/qcmain.aspx?d=102072).]*
- `PeekChar` does the same as `Read`, before moving the seek pointer back to what it was before `PeekChar` was called. *[NB: the bug that affects the parameterless Read affects this method too. As a result, if you wish to check whether the underlying stream is at its end, use Reader.BaseStream.Position in combination with Reader.BaseStream.Size instead.]*
- `ReadChar` calls `Read` and typecasts the result to `Char` if the stream wasn't at its end, otherwise an exception is raised.
- `ReadChars` reads the specified number of characters (`TCharArray` is just an alias for `TArray<Char>`). This works regardless of the encoding used — if necessary, character data will be converted to Delphi's native UTF-16.
- The version of `Read` with parameters does the same as `ReadChars`, only writing to a user-supplied buffer. Don't be confused by the `const` — this is a hint to the compiler not to bother incrementing the dynamic array's internal reference count, not a statement that `Buffer` will be left unmodified by `Read`.
- `ReadString` reads back a string that was written using `TBinaryWriter.WriteString` or a compatible method (more on which shortly).

`TBinaryWriter`'s methods look very similar, only with them defined as overloads for a single `Write` method rather than as `WriteBoolean`, `WriteByte`, etc.:

```
procedure Write(Value: Byte); overload; virtual;
procedure Write(Value: Boolean); overload; virtual;
//...
procedure Write(Value: Char); overload; virtual;
procedure Write(const Value: TCharArray); overload; virtual;
procedure Write(const Value: string); overload;
procedure Write(const Value: TCharArray; Index, Count: Integer);
```

Like `ReadString` in `TBinaryReader`, the string version of `Write` here is a special case, since it prepends the data it writes with

a length marker (none of the other `Write` methods do that). The size of the marker itself varies depending on the number of characters to write: if the source string is up to 127 characters in length, one byte is used, otherwise two bytes are. Either way, `TBinaryReader.ReadString` will understand the format, and beyond Delphi, so too will the `ReadString` method of the .NET `BinaryReader` class.

Limitations of TBinaryReader and TBinaryWriter

With the exception of their support for reading and writing strings and character data, `TBinaryReader` and `TBinaryWriter` are very simple things. Thus, they don't 'buffer' the underlying stream, offer no assistance with respect to things such as 'endianness' ('big endian' files such as JPEG structures use a different integer format than the 'little endian' one used by Delphi and Intel-targeting compilers more generally), and make no checks on the data read in or written out. As a result, the majority of `TBinaryReader`'s `ReadXXX` calls are simply implemented as

```
function TBinaryReader.ReadSomething: Something;
begin
  FStream.ReadBuffer(Result, SizeOf(Result));
end;
```

The various `Write` overloads of `TBinaryWriter` are implemented similarly, only calling `Stream.Write` or `Stream.WriteBuffer` rather than `Stream.ReadBuffer`.

`TBinaryWriter` also has a particular issue with how it defines a single `Write` method with multiple overloads instead of a series of separately-named methods (`WriteBoolean`, `WriteInt32`, etc.). While on the face of it this allows for cleaner-looking code, the way it obscures what data types are actually being used can be unhelpful in practice. For example, consider the following code:

```
procedure WriteInt32Array(const Arr: TArray<Int32>;
  Writer: TBinaryWriter);
var
  Elem: Int32;
begin
  Writer.Write(Length(Arr));
  for Elem in Arr do
    Writer.Write(Elem);
end;
```

Since the `Length` standard function will return an `Int32` when you create a 32 bit program and an `Int64` when you create a 64 bit one, this procedure will produce different output for Win32/OS X and Win64, causing misreads if you attempt to read data written using a 32 bit application in a 64 bit one or vice versa!

Ironically, using the raw `TStream` interface instead of `TBinaryWriter` would make such an error much harder to fall into, since the signature of `TStream.WriteBuffer` does not facilitate passing a function result directly to it — instead, you must explicitly declare a variable, and pass that.

Text readers and writers

`TTextReader` and `TTextWriter` are a pair of abstract classes that define a simple interface for reading and writing purely textual data, for example plain text files (*.txt, *.log, etc.). Descending from `TTextReader` and `TTextWriter`, `TStringReader` and `TStringWriter` provide concrete implementations that work on a string, and `TStreamReader` and `TStreamWriter` (which we will look at here) provide implementations that work on a stream or file:

```
uses System.SysUtils, System.Classes;

var
  LogFileName: string;
  Reader: TStreamReader;
  Writer: TStreamWriter;
begin
  //write out a log file...
  LogFileName := ChangeFileExt(GetModuleName(0), '.log');
  Writer := TStreamWriter.Create(LogFileName);
  try
    Writer.WriteLine(DateTimeToStr(Now));
    Writer.WriteLine('A handy class has been used');
  finally
    Writer.Free;
  end;
  //read it back in again, line by line...
  Reader := TStreamReader.Create(LogFileName);
  try
    while not Reader.EndOfStream do
```

```
      WriteLn(Reader.ReadLine);
  finally
    Reader.Free;
  end;
end.
```

`TTextReader` defines the following methods:

```
procedure Close; virtual; abstract;
function Peek: Integer; virtual; abstract;
function Read: Integer; overload; virtual; abstract;
function Read(const Buffer: TCharArray;
  Index, Count: Integer): Integer; overload; virtual; abstract;
function ReadBlock(const Buffer: TCharArray;
  Index, Count: Integer): Integer; virtual; abstract;
function ReadLine: string; virtual; abstract;
function ReadToEnd: string; virtual; abstract;
```

As implemented by `TStreamReader`, they do this:

- `Close` frees any owned stream and/or buffered data ahead of the reader itself being freed. (If you don't call `Close`, then the destructor will.)
- `Peek` returns the Unicode ordinal value of the next character in the underlying stream before restoring the seek pointer to where it was before. If there is no more data to read from the stream then -1 is returned, however if that is your only concern, you can just check the `EndOfStream` property instead:

```
   if Reader.EndOfStream then
     ShowMessage('No more text to read!');
```

- `Read` in its single parameter form does the same as `Peek`, only without restoring the stream's position.
- The other variant of `Read`, together with `ReadBlock` (they are equivalent) reads the specified number of characters into the provided buffer, filling it from `Index` (the first character has an index of 0). For example, if the next three characters in the stream are 'you' and the array passed in is ten characters long with the content 'me and her' immediately beforehand, calling `Read(MyArr, 7, 3)` will change its content to 'me and you'. Due to the internal implementation of strings and dynamic arrays, it is technically OK (if not semantically correct) to pass in a typecast string for `Buffer`: `ReadBlock(TCharArray(MyStr), 7, 3)`.
- `ReadLine` reads characters up until the next line ending sequence, before positioning the stream's seek pointer immediately after it. If the end of the stream is reached, then the characters to the end are returned; if it was *already* reached, an empty string is returned.

  Line endings can be just a carriage return character (`#13`), just a linefeed character (`#10`), or a carriage return followed by a linefeed character (`#13#10`), as per pre-OS X Mac, Unix, and Windows (DOS, CP/M) traditions respectively. It is acceptable for the same stream to have inconsistent line endings:

```
uses System.SysUtils, System.Classes;

const
  TestStr = 'First'#13'Second'#13#10'Third'#10'Last';
var
  Stream: TMemoryStream;
  Reader: TStreamReader;
begin
  Stream := TMemoryStream.Create;
  try
    //create a text stream with inconsistent line endings
    Stream.WriteBuffer(TestStr[1], ByteLength(TestStr));
    Stream.Position := 0;
    //get a TStreamReader to read it
    Reader := TStreamReader.Create(Stream, TEncoding.Unicode);
    try
      while not Reader.EndOfStream do
        Write(Reader.ReadLine, '/');
      //output: First/Second/Third/Last/
    finally
      Reader.Free;
    end;
  finally
    Stream.Free;
  end;
end.
```

- `ReadToEnd` reads characters from the stream's current position to its end, leaving the seek pointer there. Once again, an

empty string is returned if the end has already been reached.

Many of these methods are similar to ones provided by `TBinaryReader`, though unlike that class, `TStringReader` buffers its reads. This means it gets data from the stream in fixed-size chunks to minimise the number of times it must request new data.

The default buffer size is 1KB (1024 bytes). This however can be customised when constructing the reader. In all, `TStreamReader` declares the following constructors:

```
constructor Create(Stream: TStream);
constructor Create(Stream: TStream; DetectBOM: Boolean);
constructor Create(Stream: TStream; Encoding: TEncoding;
  DetectBOM: Boolean = False; BufferSize: Integer = 1024);
constructor Create(const Filename: string);
constructor Create(const Filename: string; DetectBOM: Boolean);
constructor Create(const Filename: string; Encoding: TEncoding;
  DetectBOM: Boolean = False; BufferSize: Integer = 1024);
```

As with `TBinaryReader`, a `TFileStream` will be created internally if you use one of the file name variants. When you pass a stream in explicitly, call the `OwnStream` method immediately after `Create` to have the reader automatically free it when the reader itself is destroyed:

```
//open a file and still allowing other processes to read it too
Reader := TStreamReader.Create(TFileStream.Create('C:\Text.txt',
  fmOpenRead or fmShareDenyWrite));
try
  Reader.OwnStream;
  //use Reader...
finally
  Reader.Free; //will free the stream too
end;
```

Again similar to `TBinaryReader`, you can retrieve the underlying stream of an active `TStreamReader` via the `BaseStream` property. The encoding can be retrieved via `CurrentEncoding`:

```
if Reader.BaseStream is TFileStream then
  WriteLn('Reading from a file');
WriteLn('The encoding used is ', Reader.CurrentEncoding.EncodingName);
```

When `DetectBOM` is either `True` or left unspecified, byte order marks (BOMs) for UTF-8, UTF-16LE and UTF-16BE are looked for and the corresponding `TEncoding` object used if found. If a BOM isn't found, then the default encoding will be used. On Windows this will be `TEncoding.Ansi`, on the Mac `TEncoding.UTF8`. When `DetectBOM` is `False`, in contrast, valid BOMs are still looked for, but simply skipped when discovered. This behaviour is particularly useful for UTF-8 encoded text files, which by convention have a BOM on Windows but not on *nix (OS X, Linux, etc.).

TStreamReader and TEncoding

When BOM auto-detection is enabled, `TStreamReader` looks for a BOM at the position of the stream's seek pointer the first time data is read using the reader object. If you create a file stream, read some data from it, then construct the reader without resetting the stream's `Position` property first (or calling `Seek(0, soBeginning)`), any BOM at the head of the file therefore won't be picked up.

If you still wish to detect a BOM in such a case, you must check for it yourself. The following code demonstrates this:

```
uses
  System.SysUtils, System.Classes;

type
  TBinaryHeader = packed record
    MajorVersion, MinorVersion: Byte;
  end;

var
  BinaryHeader: TBinaryHeader;
  Encoding: TEncoding;
  Line: string;
  PossibleBOM: TBytes;
  Stream: TFileStream;
  Reader: TStreamReader;
begin
  Stream := TFileStream.Create(SomeFile,
    fmOpenRead or fmShareDenyWrite);
  try
    SetLength(PossibleBOM, 6);
```

```
    Stream.Read(PossibleBOM[0], Length(PossibleBOM));
    { Detect any BOM, and place seek pointer immediately after
      it if found - GetBufferEncoding returns TEncoding.Default
      and 0 if there wasn't a BOM (pass a third parameter to
      have a different default encoding) }
    Encoding := nil;
    Stream.Position := TEncoding.GetBufferEncoding(PossibleBOM,
      Encoding);
    { Read some stuff independent of TStreamReader... }
    Stream.ReadBuffer(BinaryHeader, SizeOf(TBinaryHeader));
    { Construct and use TStreamReader with detected encoding }
    Reader := TStreamReader.Create(Stream, Encoding, False);
    try
      { Use Reader... }
      while not Reader.EndOfStream do
      begin
        Line := Reader.ReadLine;
        //...
      end;
    finally
      Reader.Free;
    end;
  finally
    Stream.Free;
  end;
end.
```

A separate issue concerns what happens when no BOM is found: while you can apparently specify a default encoding when BOM auto-detection is enabled, this will actually get ignored if it doesn't match `TEncoding.Default` — instead, `TEncoding.UTF8` will always be used.

The following program demonstrates this problem. In it, a BOM-less stream of UTF-16 characters is only read correctly if BOM auto-detection is explicitly disabled:

```
uses
  System.SysUtils, System.Classes, Vcl.Dialogs;

const
  SourceData: UnicodeString = 'Café Motörhead';
var
  Stream: TMemoryStream;
  Reader: TStreamReader;
begin
  Reader := nil;
  Stream := TMemoryStream.Create;
  try
    Stream.WriteBuffer(SourceData[1],
      Length(SourceData) * StringElementSize(SourceData));
    Stream.Position := 0;
    { Try with BOM auto-detection on... }
    Stream.Position := 0;
    Reader := TStreamReader.Create(Stream,
      TEncoding.Unicode, True);
    ShowMessage('With BOM auto-detection and the actual ' +
      'encoding specified as the default: ' + Reader.ReadLine);
    FreeAndNil(Reader);
    { Try with no BOM auto-detection... }
    Stream.Position := 0;
    Reader := TStreamReader.Create(Stream, TEncoding.Unicode);
    ShowMessage('With no BOM auto-detection and the actual ' +
      'encoding specified as the default: ' + Reader.ReadLine);
  finally
    Stream.Free;
    Reader.Free;
  end;
end.
```

Here, the first attempt at reading the string will either retrieve gibberish or nothing at all. The workaround is to do what the second attempt does, which is to just disable BOM auto-detection when the actual encoding is known.

TTextWriter/TStreamWriter

`TTextWriter` defines four methods, `Close`, `Flush`, `Write` and `WriteLine`. As in `TStreamReader`, `TStreamWriter` implements `Close` to free any owned stream ahead of the writer's own destruction. Also like `TStreamReader`, `TStreamWriter` buffers its operations, however by default the buffer is 'flushed' (i.e., outputted to the underlying stream) after every write. To

prevent that from happening, change the `AutoFlush` property to `False`. When done, you can call `Flush` at your discretion (it is called automatically by both `Close` and the destructor, and when the buffer gets full).

`Write` and `WriteLine` are both overloaded to accept either a `string` directly, or a `Boolean`, `Char`, `Single`, `Double`, `Integer`/`Int32`, `Cardinal`/`UInt32`, `Int64` or `UInt64`. In the case of a number, a string representation of it rather than the number itself is written out. Also available are overloads that take a `TObject` descendant; in their case, the object's `ToString` method is called and the result streamed out. By default that won't output anything very interesting — just the class name for most objects — however it can be customised given `ToString` is virtual.

For convenience, there are also overloads of `Write` and `WriteLine` that take a string and an `array of const`. These just call the `Format` function and send the result on to the single-parameter version of `Write` or `WriteLine`:

```
//the next two lines are equivalent
Writer.Write('This is a number: %d', [42]);
Writer.Write(Format('This is a number: %d', [42]));
```

The new line sequence written out by `WriteLine` is controlled by the `NewLine` property. By default this is `#13#10` on Windows and `#10` on the Mac. In principle you can set it to whatever string you want though, however it would be confusing to have `WriteLine` not actually output a line break of any description.

`TStreamWriter`'s constructors look like this:

```
constructor Create(Stream: TStream);
constructor Create(Stream: TStream; Encoding: TEncoding;
  BufferSize: Integer = 1024);
constructor Create(const Filename: string;
  Append: Boolean = False);
constructor Create(const Filename: string; Append: Boolean;
  Encoding: TEncoding; BufferSize: Integer = 1024);
```

When a `TEncoding` object is explicitly specified, any BOM it defines will be written out *if* the stream's `Position` property returns 0 on the writer being created. This behaviour does not correspond to `TStreamReader`'s, which checks for a BOM at the stream's current position, not its head.

If you don't explicitly specify an encoding, then `TEncoding.UTF8` is chosen for you, and *no* BOM outputted. If you want to use UTF-8 and still output a BOM, you must therefore pass the encoding object explicitly:

```
Writer := TStreamWriter.Create('C:\Users\Bob\Has BOM.txt',
  False, TEncoding.UTF8);
```

Either way, once created, you can check the `Encoding` property to see what encoding is being used.

TReader and TWriter

Where `TStreamReader` and `TStreamWriter` treat their underlying streams as streams of text, `TReader` and `TWriter` are more like `TBinaryReader` and `TBinaryWriter` in thinking in terms of binary data. However, unlike `TBinaryWriter`, `TWriter` writes type markers, which `TReader` then looks for. Thus, if you write an `Int32` using `TBinaryWriter`, it literally just writes out the `Int32` value. In contrast, `TWriter` will write an additional byte that says, in effect, 'an `Int32` now follows'. Then, when you ask `TReader` to read an `Int32`, it checks for this marker and raises an exception if it can't find it.

On the one hand, this gives `TReader` and `TWriter` an element of type safety that is inherently missing from `TBinaryXXX`. On the other hand though, it makes them more Delphi-specific. Put another way, `TReader` isn't for reading *any* file, just files written using `TWriter`. Nonetheless, the format is stable — `TReader` and `TWriter` were originally written to support streaming VCL form definition files (DFMs), and both backwards and forwards compatibility is such that files written using `TWriter` in XE2 can for the most part be read using Delphi 1's `TReader`.

Using TReader and TWriter

Like `TStreamReader` and `TStreamWriter`, `TReader` and `TWriter` buffer the data they read and write from their underlying stream. However, you must explicitly specify the buffer size on creation, where 500 will be 500 bytes, 1024 ($400 in hexadecimal notation) will be 1KB, and so on:

```
Reader := TReader.Create(StreamToReadFrom, $1800); //6KB buffer
Writer := TWriter.Create(StreamToWriteTo, 1024);   //1KB buffer
```

Neither constructor is overloaded — you simply pass the underlying stream and desired buffer size. This means you aren't able to specify a `TEncoding` instance to use for string data. Instead, `TWriter` decides the encoding for you on a string-by-string basis, choosing UTF-8 if that would produce a byte length less than UTF-16, otherwise UTF-16 itself is used.

Along with a similar constructor, `TReader` and `TWriter` share a `Position` property and a `FlushBuffer` method. The first

returns or sets the underlying stream's similarly-named property in a manner that adjusts for the reader or writer's internal buffer. In the case of `TReader`, `FlushBuffer` clears whatever is currently buffered (cached), causing the next substantive call to cause the reader to go back to the stream for data; `TWriter`, in contrast, implements `FlushBuffer` by outputting whatever has been buffered up to that point.

As you would expect, the bulk of `TReader`'s public interface is taken by `ReadXXX` methods, whose range is similar if not quite the same as those declared by `TBinaryReader`. Each corresponds to a `WriteXXX` method on `TWriter`:

- `TReader.ReadChar` and `TWriter.WriteChar` are implemented internally as reading and writing a string with one element. This makes `TReader.ReadChar` compatible with single character strings written with `WriteString`.
- `TReader.ReadCollection` and `TWriter.WriteCollection` are for reading and writing a `TCollection` descendant, i.e. a streamable list (`TCollection` will be discussed later on in this chapter).
- `TReader.ReadComponent` and `TWriter.WriteComponent` read and write a `TComponent` descendant's 'published' properties (i.e., properties with `published` visibility).
- `TReader.ReadCurrency` and `TWriter.WriteCurrency` read and write a `Currency` value. It is acceptable to call `TReader.ReadCurrency` even when `WriteInteger` was used to write the value out in the first place.
- `TReader.ReadDate` and `TWriter.WriteDate` read and write a `TDateTime` value. Despite the naming, neither chop off the time part.
- `TReader.ReadDouble` and `TWriter.WriteDouble` read and write a `Double` value. `ReadDouble` may be called when `WriteInteger` was used to write the value, however if `WriteSingle` was, an exception will be raised.
- `TReader.ReadFloat` and `TWriter.WriteFloat` read and write `Extended` values, and in a way that handles how this type has a different size when compiling for 32 or 64 bits.
- `TReader.ReadInteger`, `TReader.ReadInt64` and `TWriter.WriteInteger` read and write integers. `WriteInteger` is overloaded to accept either an `Integer` or an `Int64`; in either case, a signed integer of the smallest size able to store the value will be written out. In its current implementation, `ReadInteger` will raise an exception if a 64 bit integer was written, otherwise both it and `ReadInt64` can read any integer written using `WriteInteger`.
- `TReader.ReadSingle` and `TWriter.WriteSingle` read and write `Single` values; `ReadSingle` may be used even if `WriteInteger` rather than `WriteSingle` was used to write the value.
- `TReader.ReadString` and `TWriter.WriteString` read and write strings. Do not confuse these methods with `ReadStr`/`WriteStr` and `ReadUTF8Str`/`WriteUTF8Str`, which are lower-level routines that don't output a type kind marker.
- `TReader.ReadVariant` and `TWriter.WriteVariant` read and write variants. Variant arrays are not supported, however custom variants will be if they implement the `IVarStreamable` interface.

The type markers written out by the `WriteXXX` methods are represented by the following enumeration:

```
type
  TValueType = (vaNull, vaList, vaInt8, vaInt16, vaInt32,
    vaExtended, vaString, vaIdent, vaFalse, vaTrue, vaBinary,
    vaSet, vaLString, vaNil, vaCollection, vaSingle, vaCurrency,
    vaDate, vaWString, vaInt64, vaUTF8String, vaDouble);
```

`TReader` defines `ReadValue` and `NextValue` methods to read the marker directly; where `ReadValue` leaves the seek pointer immediately after the marker, `NextValue` rolls it back. In both cases an exception will be raised if the stream is already at its end.

One way to avoid an exception is to check whether the `Position` property of the reader is less than the `Size` property of the stream. However, a better way is to use `TReader` and `TWriter`'s built-in (if basic) support for open-ended lists of values. When writing, this works by simply outputting items in between `WriteListBegin` and `WriteListEnd` calls; when reading the data back in, call `ReadListBegin`, then the `EndOfList Boolean` function in the context of a while loop, before `ReadListEnd` to finish things off. The following example does this in order to stream a generic list of `string`:

```
uses
  System.SysUtils, System.Classes, System.Generics.Collections;

procedure WriteStrings(Source: TList<string>; Writer: TWriter);
var
  Item: string;
begin
  Writer.WriteListBegin;
  for Item in Source do
```

```
    Writer.WriteString(Item);
  Writer.WriteListEnd;
end;

procedure ReadStrings(Dest: TList<string>; Reader: TReader);
var
  Item: string;
begin
  Reader.ReadListBegin;
  while not Reader.EndOfList do
    Dest.Add(Reader.ReadString);
  Reader.ReadListEnd;
end;
```

Note that `XxxListBegin` and `XxxListEnd` calls can be nested. However, each `XxxListBegin` much have a matching `XxxListEnd`.

Component streaming

`TStream` does not support streaming any given object *en bloc*. However, it does have built-in support for components. This is intricately bound up with `TComponent` itself, and was originally geared towards the streaming needs of the VCL and FireMonkey. Nonetheless, it can in principle be used for other purposes too, and in that mode, forms the functional equivalent of object 'archiving' in Objective-C/Cocoa and object 'serialisation' in Java or .NET, albeit with certain restrictions.

Using the ReadComponent and WriteComponent methods of TStream

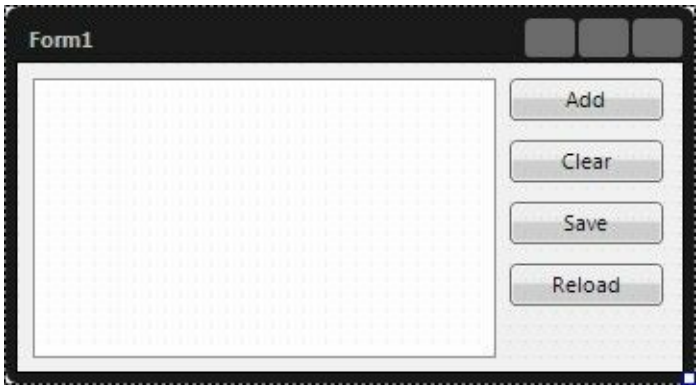
The basic API for component streaming takes the form of two methods on `TStream`, `ReadComponent` and `WriteComponent`:

```
function ReadComponent(Instance: TComponent): TComponent;
procedure WriteComponent(Instance: TComponent);
```

Here, `Instance` is said to be the 'root component'. By default, `WriteComponent` streams all the root component's properties that have `published` visibility, together with the published properties of all components returned by the root's `GetChildren` virtual method. In the case of a control or form (FireMonkey or VCL), this means the root's child controls, which in turn will get their own children streamed and so on.

As you would expect, calling `ReadComponent` has everything loaded back in; if `Instance` is `nil`, then the root component itself will be created too. If there are child components involved, `ReadComponent` will construct them afresh. Because of that, if you wish to avoid duplicating components, you must free any old ones before calling `ReadComponent`. In the case of components added to a form or data module at design time, you *must* do this on pain of causing runtime exceptions: since components with the same owner must either be uniquely named or not be named at all, and all components added at design-time *are* named, name clashes will arise if you don't free existing children.

To see how `WriteComponent` and `ReadComponent` work in practice, let us write a little demo. So, first create a new FireMonkey application, and add to the form a `TFramedScrollBox` and four `TButton` controls. Name these `scbHost`, `btnAdd`, `btnClear`, `btnSave` and `btnReload` respectively, and arrange them to look like this:



Next, handle the first button's `OnClick` event as thus:

```
procedure TForm1.btnAddObjectClick(Sender: TObject);
var
  NewObject: TSelection;
begin
  NewObject := TSelection.Create(scbHost);
  NewObject.Position.X := 20 * scbHost.ComponentCount;
  NewObject.Position.Y := 20 * scbHost.ComponentCount;
  NewObject.Parent := scbHost;
end;
```

This creates a new `TSelection` control and adds it to the scroll box (you will need to ensure `FMX.Objects` is included in the unit's `uses` clause for this to compile, since it is there that `TSelection` is defined). Next, handle `btnClear`'s `OnClick` event like this:

```
procedure TForm1.btnClearClick(Sender: TObject);
var
  I: Integer;
begin
  for I := scbHost.ComponentCount - 1 downto 0 do
    if scbHost.Components[I] is TSelection then
      scbHost.Components[I].Free;
end;
```

This simply deletes any `TSelection` objects created previously. Finally, handle the other two buttons' `OnClick` events like

so:

```
function GetLayoutFileName: string;
begin
  Result := ChangeFileExt(GetModuleName(0), '.layout');
end;

procedure TForm1.btnSave(Sender: TObject);
var
  Stream: TFileStream;
begin
  Stream := TFileStream.Create(GetLayoutFileName, fmCreate);
  try
    Stream.WriteComponent(scbHost);
  finally
    Stream.Free;
  end;
end;

procedure TForm1.btnReload(Sender: TObject);
var
  Stream: TFileStream;
begin
  Stream := TFileStream.Create(GetLayoutFileName, fmOpenRead);
  try
    Stream.ReadComponent(scbHost);
  finally
    Stream.Free;
  end;
end;
```

For simplicity, we just save and restore to a file with a `.layout` extension, named after the executable, and placed in the executable's folder.

If you run the application, you should be able to add a series of objects to the scroll box, reposition and resize them as you wish, then save, clear and restore them. If you don't clear the old ones before reloading, components will be duplicated. Since we don't explicitly assign the `Name` property of the components we construct at runtime, this won't be an error — just potentially a bit confusing for the user.

Going deeper

Available in this book's source code repository is a slightly fuller version of the example just presented. In the second variant, pictures are dropped into the scroll box rather than just selection rectangles:

While on the same lines as the original demo, this version goes a bit deeper into the component streaming system. In particular, a custom control type is used, which requires registration with the system. Furthermore, the ability to save using a human-readable text format rather than the default binary format is implemented. Let's look at the mechanics of both these features.

Custom components

In itself, `TStream.WriteComponent` will work with whatever component tree you pass it; similarly, `TStream.ReadComponent` will happily accept any sort of root component. However, `ReadComponent` may require you explicitly register potential child component types first. In the case of custom FireMonkey controls, this means calling `RegisterFmxClasses` (declared in `FMX.Types`); otherwise, call either `RegisterClass` or `RegisterClasses`, both of which are declared in `System.Classes` and should not be confused with `RegisterComponents`, which is for registering with the IDE. A class neutral between the FireMonkey and VCL GUI frameworks should be registered with `RegisterClass`/`RegisterClasses`.

Whichever registration routine used, normal practice is to call it in the `initialization` section of a unit:

```
initialization
  RegisterFmxClasses([TMyCustomFMXControl]);
  RegisterClasses([TMyNeutralObject]);
end.
```

Standard FireMonkey control types are automatically registered by virtue of their units calling `RegisterFmxClasses` for you. Most standard VCL control types are *not* however.

Nonetheless, this last point can be obscured by the fact explicit registration of VCL classes is not necessary in one particular case: streaming in a VCL form that has exactly the same controls at runtime as it had at design-time. The reason for that is because the streaming system checks the 'runtime type information' (RTTI) generated for the published fields of the root component *in addition* to the registered class list. Since controls added at design-time each get a published field, this can make form streaming appear more 'magic' than it actually is.

In reality, relying upon the published field lookup is a brittle way of going about things — just call `RegisterClasses` and be done with it. If you have many possible classes to register, you can do it generically by polling the 'extended' RTTI system for them:

```
uses System.Rtti, Vcl.Controls;

procedure RegisterAllVCLControls;
var
  Context: TRttiContext;
  LType: TRttiType;
  Metaclass: TClass;
begin
  for LType in Context.GetTypes do
    if LType.IsInstance then
    begin
      Metaclass := TRttiInstanceType(LType).MetaclassType;
      if Metaclass.InheritsFrom(TControl) then
        RegisterClass(TControlClass(Metaclass));
    end;
end;
```

Binary vs. textual formats

If you open up a file generated by `TStream.WriteComponent`, you will find data has been persisted in a binary format that you can't easily edit. If you open up a Delphi form definition file however (.dfm or .fmx), you will usually discover a human-readable textual format used instead. For example, a simple FireMonkey form that contains a label, edit box and button might be streamed as the following:

```
object Form1: TForm1
  Left = 0
  Top = 0
  Caption = 'Form1'
  ClientHeight = 61
  ClientWidth = 350
  Visible = False
  StyleLookup = 'backgroundstyle'
  object Edit1: TEdit
    Position.Point = '(8,24)'
    Width = 241.000000000000000000
    Height = 22.000000000000000000
    TabOrder = 1
```

```
    KeyboardType = vktDefault
    Password = False
    object Label1: TLabel
      Position.Point = '(0,-16)'
      Width = 120.000000000000000000
      Height = 15.000000000000000000
      TabOrder = 1
      Text = 'Enter your name:'
    end
  end
  object Button1: TButton
    Position.Point = '(256,24)'
    Width = 80.000000000000000000
    Height = 22.000000000000000000
    TabOrder = 10
    Text = 'OK'
  end
end
```

Just as the component streaming system generally is available to use, so too is this textual format. Specifically, it is available via the following routines, all of which are declared in the `System.Classes` unit:

```
type
  TStreamOriginalFormat = (sofUnknown, sofBinary, sofText,
    sofUTF8Text);

function TestStreamFormat(Stream: TStream): TStreamOriginalFormat;

procedure ObjectBinaryToText(Input, Output: TStream); overload;
procedure ObjectBinaryToText(Input, Output: TStream;
  var OriginalFormat: TStreamOriginalFormat); overload;
procedure ObjectTextToBinary(Input, Output: TStream); overload;
procedure ObjectTextToBinary(Input, Output: TStream;
  var OriginalFormat: TStreamOriginalFormat); overload;
```

In order to output a file that uses the textual format, call `WriteComponent` to output to a temporary stream, then `ObjectBinaryToText` to convert from the binary format to the textual one:

```
procedure TExampleForm.SaveToFile(const AFileName: string;
  AsText: Boolean);
var
  FileStream: TFileStream;
  MemStream: TMemoryStream;
begin
  MemStream := nil;
  FileStream := TFileStream.Create(AFileName, fmCreate);
  try
    if not AsText then
      FileStream.WriteComponent(scbHost)
    else
    begin
      MemStream := TMemoryStream.Create;
      MemStream.WriteComponent(scbHost);
      MemStream.Seek(0, soBeginning);
      ObjectBinaryToText(MemStream, FileStream);
    end;
  finally
    FileStream.Free;
    MemStream.Free;
  end;
end;
```

To go the other way in a fashion that supports both formats, first call `TestStreamFormat` to determine which format was used. If `sofBinary` is returned, call `ReadComponent` directly, otherwise call `ObjectTextToBinary` then `ReadComponent`:

```
procedure TExampleForm.LoadFromFile(const AFileName: string);
var
  FileStream: TFileStream;
  MemStream: TMemoryStream;
begin
  MemStream := nil;
  FileStream := TFileStream.Create(AFileName, fmOpenRead);
  try
    if TestStreamFormat(FileStream) = sofBinary then
      FileStream.ReadComponent(scbHost)
    else
    begin
      MemStream := TMemoryStream.Create;
```

```
      ObjectTextToBinary(FileStream, MemStream);
      MemStream.Seek(0, soBeginning);
      MemStream.ReadComponent(scbHost)
    end;
  finally
    FileStream.Free;
    MemStream.Free;
  end;
end;
```

The cost of using the textual format is that file sizes will be bigger, especially if the streamed out properties include intrinsically binary data (for example graphics). Nonetheless, the benefits of using a format that allows for manual editing will frequently outweigh this cost.

Streaming things beyond visual controls

Component streaming was designed for streaming controls, and in particular, Delphi forms. If certain requirements are adhered to however, it can in principle be used much more generally. The main limitations are the following:

- Root objects must descend from `TComponent` or a descendant of that class.
- To be streamed, a property must have `published` visibility *and* be read/write.
- Array properties (`property MyProp[Index: SomeType]: SomeType read ...`) are not streamable at all, along with properties that have a pointer, record or array type. The compiler enforces this by not allowing such properties to have `published` visibility in the first place.
- Object properties specifically must be typed to classes descended from `TPersistent`. If the class descends from `TComponent`, then the sub-object must have its `SetSubComponent` method called immediately after being constructed:

```
  type
    TChildComponent = class(TComponent)
    //...
    end;

    TSomeRootComponent = class(TComponent)
    strict private
      FChild: TChildComponent;
      procedure SetChild(AValue: TComponent);
    published
      property Child: TChildComponent read FChild write SetChild;
    end;

  constructor TSomeRootComponent.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);
    FChildComp := TChildComponent.Create(Self);
    FChildComp.SetSubComponent(True);
    //...
  end;

  procedure TSomeRootComponent.SetChild(AValue: TComponent);
  begin
    FChildComp.Assign(AValue);
  end;
```

  If you don't call `SetSubComponent`, the streaming system will assume the property defines a link to an independent component, rather than to an object whose lifetime is managed by the root component itself.
- Any relevant piece of state not exposed by a streamable property must be persisted by overriding the `DefineProperties` method introduced in `TPersistent`. For example, both the VCL and FireMonkey versions of the `TBitmap` class override `DefineProperties` to persist image data.

Let's look at an example of how these limitations work in practice. Consider the following code:

```
uses
  System.SysUtils, System.Classes, System.Generics.Collections;

type
  TPerson = class
  strict private
    FSurname: string;
    FForename: string;
  public
    property Forename: string read FForename write FForename;
```

```
    property Surname: string read FSurname write FSurname;
  end;

  TVisitData = class(TComponent)
  strict private
    FDescription: string;
    FPeople: TObjectList<TPerson>;
    FPlaces: TList<string>;
  public
    constructor Create(AOwner: TComponent = nil); override;
    destructor Destroy; override;
    property Description: string read FDescription
      write FDescription;
    property People: TObjectList<TPerson> read FPeople;
    property Places: TList<string> read FPlaces;
  end;

constructor TVisitData.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPeople := TObjectList<TPerson>.Create;
  FPlaces := TList<string>.Create;
end;

destructor TVisitData.Destroy;
begin
  FPlaces.Free;
  FPeople.Free;
  inherited;
end;
```

While it descends from `TComponent`, `TVisitData` isn't stream friendly for a number of reasons: properties aren't published, some properties don't have setters, and generic list classes (which don't descend from `TPersistent`) are used.

Fixing some of these things is easy: add published sections, add setters, and for the `Places` property, replace `TList<string>` with the `TPersistent`-deriving `TStrings`/`TStringList`:

```
uses
  System.SysUtils, System.Classes, System.Generics.Collections;

type
  TPerson = class
  //... as before
  end;

  TVisitData = class(TComponent)
  strict private
    FDescription: string;
    FPeople: TObjectList<TPerson>;
    FPlaces:TStrings;
    procedure SetPlaces(AValue: TStrings);
  public
    constructor Create(AOwner: TComponent = nil); override;
    destructor Destroy; override;
    //nb: the People property is still to fix!
    property People: TObjectList<TPerson> read FPeople;
  published
    property Description: string read FDescription
      write FDescription;
    property Places: TStrings read FPlaces write SetPlaces;
  end;

constructor TVisitData.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPeople := TObjectList<TPerson>.Create;
  FPlaces := TStringList.Create;
end;

destructor TVisitData.Destroy;
begin
  FPlaces.Free;
  FPeople.Free;
  inherited;
end;

procedure TVisitData.SetPlaces(AValue: TStrings);
```

```
begin
  FPlaces.Assign(AValue);
end;
```

This leaves dealing with the `People` property. There is more than one possible solution for it, though two in particular stand out: having the root component override `DefineProperties` to write the list out manually, and converting the list to a streamable one so it can persist itself. Let's look at both solutions in practice.

Implementing DefineProperties

Keeping the `TPerson` class and the `People` property exactly as they were, add the following methods to `TVisitData`:

```
  strict private
    procedure ReadPeople(Reader: TReader);
    procedure WritePeople(Writer: TWriter);
  protected
    procedure DefineProperties(Filer: TFiler); override;
```

Implement them like this:

```
procedure TVisitData.DefineProperties(Filer: TFiler);
begin
  inherited;
  Filer.DefineProperty('People', ReadPeople, WritePeople,
    FPeople.Count > 0);
end;

procedure TVisitData.ReadPeople(Reader: TReader);
var
  NewPerson: TPerson;
begin
  Reader.ReadListBegin;
  FPeople.Clear;
  while not Reader.EndOfList do
  begin
    NewPerson := TPerson.Create;
    try
      NewPerson.Forename := Reader.ReadString;
      NewPerson.Surname := Reader.ReadString;
      FPeople.Add(NewPerson);
    except
      NewPerson.Free;
      raise;
    end;
  end;
  Reader.ReadListEnd;
end;

procedure TVisitData.WritePeople(Writer: TWriter);
var
  Person: TPerson;
begin
  Writer.WriteListBegin;
  for Person in FPeople do
  begin
    Writer.WriteString(Person.Forename);
    Writer.WriteString(Person.Surname);
  end;
  Writer.WriteListEnd;
end;
```

In implementing `DefineProperties`, we have a choice of defining callbacks that take `TReader` and `TWriter` parameters, and callbacks that take `TStream` ones. To persist its image data, the FireMonkey `TBitmap` class takes the latter option, calling as a result `Filer.DefineBinaryProperty` rather than `Filer.DefineProperty`. This is because the externally-defined format it employs (PNG) has no use for the `TReader` and `TWriter` interface. In our case `TReader` and `TWriter` makes our life easier though, so we'll use them.

With these changes made, `TVisitData` will now be fully streamable:

```
var
  Person: TPerson;
  Place: string;
  Root: TVisitData;
  Stream: TMemoryStream;
begin
  Stream := nil;
```

```
  Root := TVisitData.Create;
  try
    Root.Description := 'This should now work!';
    Root.Places.Add('Dorset');
    Root.Places.Add('Somerset');
    Person := TPerson.Create;
    Person.Forename := 'John';
    Person.Surname := 'Smith';
    Root.People.Add(Person);
    //save root object to the stream
    Stream := TMemoryStream.Create;
    Stream.WriteComponent(Root);
    //recreate root object afresh, and load it from the stream
    FreeAndNil(Root);
    Root := TVisitData.Create;
    Stream.Position := 0;
    Stream.ReadComponent(Root);
    WriteLn('Description: ', Root.Description);
    WriteLn('No. of people: ', Root.People.Count);
    if Root.People.Count > 0 then
      for Person in Root.People do
        WriteLn('  ' + Person.Forename + ' ' + Person.Surname);
    WriteLn('No. of places: ', Root.Places.Count);
    if Root.Places.Count > 0 then
      for Place in Root.Places do
        WriteLn('  ' + Place);
  finally
    Root.Free;
    Stream.Free;
  end;
end.
```

A limitation of this solution is its maintainability however. Specifically, if the properties on `TPerson` were to change, the `ReadPeople` and `WritePeople` methods of `TVisitData` would have to change in step. In contrast, the main alternative to overriding `DefineProperties` — implementing a streamable list class to replace the generic list — avoids this, albeit with the cost of a bit more coding up front.

Implementing a streamable list class (TCollection descendant)

A streamable list class is one descended from `TCollection`, typically `TOwnedCollection` in practice. Since `TCollection` implements a list of `TCollectionItem` descendants, we must change `TPerson` accordingly to descend from `TCollectionItem`. Once done however, raising its properties to `published` visibility will have them automatically streamed.

Writing the `TCollection` descendant itself takes a few steps. In essence, the task is first to let the parent class know what `TCollectionItem` descendant to use, then secondly overlay the inherited `Add` and indexer methods, which are typed to `TCollectionItem`, with ones typed to our descendant. In return, `TCollection` will do the actual list management and implement things like `Assign` for us. Here's the necessary interface:

```
type
  TPeopleList = class;

  TPerson = class(TCollectionItem)
  private
    FSurname: string;
    FForename: string;
  published
    property Forename: string read FForename write FForename;
    property Surname: string read FSurname write FSurname;
  end;

  TPeopleList = class(TOwnedCollection)
  public type
    TEnumerator = record
    strict private
      FList: TPeopleList;
      FIndex: Integer;
      function GetCurrent: TPerson;
    public
      constructor Create(AList: TPeopleList);
      function MoveNext: Boolean;
      property Current: TPerson read GetCurrent;
    end;
  strict private
    function GetItem(Index: Integer): TPerson;
```

```
  public
    constructor Create(AOwner: TPersistent);
    function GetEnumerator: TEnumerator;
    function Add: TPerson; inline;
    property Items[Index: Integer]: TPerson read GetItem; default;
  end;
```

If you don't mind somewhat broken `for`/`in` support, you can get away with not providing an enumerator, but we'll define one here for completeness' sake. It can be implemented like this:

```
constructor TPeopleList.TEnumerator.Create(AList: TPeopleList);
begin
  FList := AList;
  FIndex := -1;
end;

function TPeopleList.TEnumerator.GetCurrent: TPerson;
begin
  Result := FList[FIndex];
end;

function TPeopleList.TEnumerator.MoveNext: Boolean;
begin
  Result := (Succ(FIndex) < FList.Count);
  if Result then Inc(FIndex);
end;
```

`TPeopleList` itself can look as thus:

```
constructor TPeopleList.Create(AOwner: TPersistent);
begin
  inherited Create(AOwner, TPerson);
end;

function TPeopleList.GetEnumerator: TEnumerator;
begin
  Result := TEnumerator.Create(Self);
end;

function TPeopleList.GetItem(Index: Integer): TPerson;
begin
  Result := TPerson(inherited GetItem(Index));
end;

function TPeopleList.Add: TPerson;
begin
  Result := TPerson(inherited Add);
end;
```

Use of `TPeopleList` will require a few changes to `TVisitData`: aside from the type change, the `People` property should be both raised to `published` visibility and given a setter. This is similar to what was done for the `Places` property to help make it streamable:

```
type
  TVisitData = class(TComponent)
  strict private
    FDescription: string;
    FPeople: TPeopleList;
    FPlaces: TStrings;
    procedure SetPeople(AValue: TPeopleList);
    procedure SetPlaces(AValue: TStrings);
  public
    constructor Create(AOwner: TComponent = nil); override;
    destructor Destroy; override;
  published
    property Description: string read FDescription
      write FDescription;
    property People: TPeopleList read FPeople write SetPeople;
    property Places: TStrings read FPlaces write SetPlaces;
  end;

procedure TVisitData.SetPeople(AValue: TPeopleList);
begin
  FPeople.Assign(AValue);
end;
```

Code that uses `TVisitData` will also need to be tweaked. This is because the semantics of the `Add` method have changed:

where the `TObjectList` version *accepts* a newly-created item, the `TPeopleList` version *returns* one:

```
//old code
Person := TPerson.Create;
Person.Forename := 'John';
Person.Surname := 'Smith';
Root.People.Add(Person);

//revised code
Person := Root.People.Add;
Person.Forename := 'John';
Person.Surname := 'Smith';
```

Apart from that however, the various versions of `TVisitData` are used in exactly the same way.

Proxy streams

A 'proxy' stream class (sometimes called a 'filter') is a `TStream` descendant that performs some sort of operation on the bytes that are read from, or written to, an underlying stream. The idea of a proxy is similar to that of a reader or writer class. However, where a reader or writer's aim is to present a higher level interface from the merely byte-centric one of `TStream`, a proxy stream is itself a `TStream` descendant.

The virtue of implementing a proxy stream is that client code need not know it is reading or writing data in a specific way; rather, it just reads or writes a `TStream` instance as before. This result could be achieved instead by descending from a standard stream type; for example, buffering support might be added to `TFileStream` by creating a `TBufferedFileStream` descendant class. However, what if you later need to buffer the reads or writes of another sort of stream? By implementing buffering functionality as a proxy stream class instead of a `TFileStream` descendant, that could be achieved with no extra code.

In general, proxy stream classes are things you write yourself. However, two examples are found in the RTL, `TZCompressionStream` and `TZDecompressionStream`. Both are declared by the `System.Zlib` unit, and as their names imply, implement a compression scheme. Ultimately they form the basis of the `TZipFile` class (`TZipFile` itself will be looked at in the next chapter), however they can be used independently too. As a broader introduction to the idea of proxy streams, we will therefore look at them for their own sakes here. Following that, the mechanics of actually writing proxy streams will be looked into via a few examples — a proxy that logs information about reads, writes and seeks, and a pair of proxies that implement a generalised buffering system.

TZCompressionStream and TZDecompressionStream (ZLib unit)

The `TZCompressionStream` and `TZDecompressionStream` classes are wrappers for the Zlib compression library (`http://zlib.net/`). When targeting Windows, object files compiled from the original C source are linked into your application; when compiling for the Mac, the operating system's pre-built Zlib dynamic library is used. In either case, the interface is the same. Beyond the standard `TStream` members, it looks like this:

```
type
  TZCompressionLevel = (zcNone, zcFastest, zcDefault, zcMax);

  TZCompressionStream = class(TCustomZStream)
  public
    constructor Create(Dest: TStream); overload;
    constructor Create(Dest: TStream;
      CompressionLevel: TZCompressionLevel;
      WindowBits: Integer); overload;
    property CompressionRate: Single read GetCompressionRate;
    property OnProgress: TNotifyEvent read
      FOnProgress write FOnProgress;
  end;

  TZDecompressionStream = class(TCustomZStream)
  public
    constructor Create(Source: TStream); overload;
    constructor Create(Source: TStream; WindowBits: Integer); overload;
    property OnProgress: TNotifyEvent read
      FOnProgress write FOnProgress;
  end;
```

Attempting to read from a `TZCompressionStream` is an error; similarly, attempting to write to a `TZDecompressionStream` is also an error: both classes are strictly 'one way' in operation. When constructing the former, the speed penalty of using `zcMax` is usually not grave enough to forgo the benefits of the greater compression. If you specify it, use 15 for `WindowBits`. This determines the base two logarithm of the maximum internal buffer size, and has a valid range of 8 to 15 only (see the Zlib documentation for more details — `http://zlib.net/manual.html`):

```
procedure CompressFile(const ASourceName, ADestName: string);
var
  Source, Dest: TFileStream;
  Proxy: TZCompressionStream;
begin
  Dest := nil;
  Proxy := nil;
  Source := TFileStream.Create(ASourceName, fmOpenRead);
  try
    Dest := TFileStream.Create(ADestName, fmCreate);
    Proxy := TZCompressionStream.Create(Dest, zcMax, 15);
    Proxy.CopyFrom(Source, 0);
```

```
  finally
    Proxy.Free; //free the proxy before the base stream!
    Dest.Free;
    Source.Free;
  end;
end;
```

While Zlib implements the most popular compression method for a ZIP file, the sort of code demonstrated here does not in itself produce a a ZIP file, since ZIP header structures aren't outputted. For purely internal usage scenarios that won't matter though.

Using `TZDecompressionStream` follows a similar pattern to using `TZCompressionStream`:

```
procedure DecompressFile(const ASourceName, ADestName: string);
var
  Source, Dest: TFileStream;
  Proxy: TZDecompressionStream;
begin
  Dest := nil;
  Proxy := nil;
  Source := TFileStream.Create(ASourceName, fmOpenRead);
  try
    Proxy := TZDecompressionStream.Create(Source);
    Dest := TFileStream.Create(ADestName, fmCreate);
    Dest.CopyFrom(Proxy, 0);
  finally
    Proxy.Free;
    Dest.Free;
    Source.Free;
  end;
end;
```

If the source hasn't been compressed beforehand, an `EZDecompressionError` exception will be raised *on the first read*. Thus, if `ASourceName` does not refer to a file previously compressed with Zlib, the exception will be raised in the `CopyFrom` call, not `TZDecompressionStream.Create`.

For lengthy operations, both `TZCompressionStream` and `TZDecompressionStream` support assigning a callback to be notified of progress. Make use of this by declaring a method with a single parameter typed to `TObject`. In the method body, inspect the `Position` property of the proxy stream to learn how many uncompressed bytes of the source or destination stream have been read or written:

```
type
  TForm1 = class(TForm)
    StatusBar: TStatusBar;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  strict private
    //declare event handler
    procedure CompressionProgress(Sender: TObject);
  end;

//...

uses System.ZLib;

procedure TForm1.Button1Click(Sender: TObject);
var
  Dest: TStream;
  Proxy: TZCompressionStream;
begin
  //...
  Proxy := TCompressionStream.Create(Dest);
  try
    Proxy.OnProgress := CompressionProgress;
    //write to Proxy...
  finally
    Proxy.Free;
  end;
  StatusBar.SimpleText := 'Finished writing compressed data';
end;

procedure TForm1.CompressionProgress(Sender: TObject);
begin
  StatusBar.SimpleText := Format(
    '%d bytes written so far...', [(Sender as TStream).Position]);
```

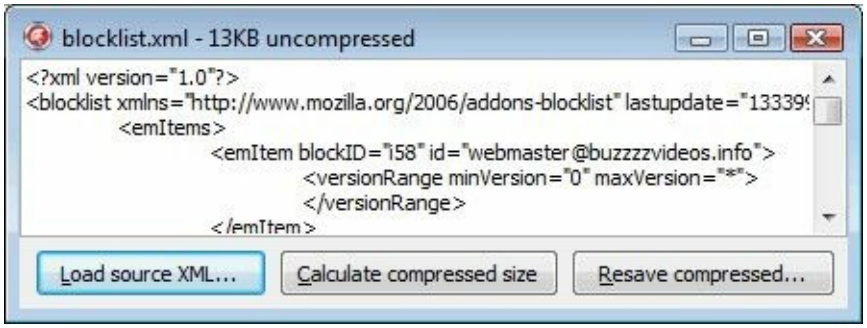
```
end;
```

By necessity, `TZCompressionStream` and `TZDecompressionStream` work by compressing and decompressing bytes in batches. In the case of the former, the `OnProgress` event is therefore raised when it flushes its internal buffer to the destination stream; in the case of the latter, after it has read the next batch of bytes from the source stream. In neither case is the event handler called 'asynchronously'. This means no new data is compressed or decompressed while the event handler is executing — rather, the proxy stream waits for your event handler to finish.

TZCompressionStream and TZDecompressionStream limitations

Aside from the fact a `TZCompressionStream` is write-only and `TZDecompressionStream` read-only, a further limitation is that both have limited `Seek` implementations. In essence, these are designed to support reading the `Position` property and little else, which can cause difficulties for the unwary — not unreasonably, much code that reads or writes to a stream expects a fully working `Seek` implementation, given the method is introduced by `TStream` itself!

For example, in this book's sample projects is a simple application that compresses XML files using `System.Zlib`:



There are two versions of the demo, one VCL and one FireMonkey. While most XML files are UTF-encoded, they need not be, and in order to avoid having to figure out the encoding itself, the demo leverages XML functionality provided by the Delphi RTL:

```
uses Xml.XMLDOM, System.ZLib;

procedure TfrmXML.LoadXMLStream(Stream: TStream);
var
  DOMDoc: IDOMDocument;
begin
  { Use the RTL's DOM support so we don't have to
    bother with encoding issues ourselves. }
  DOMDoc := GetDOM.createDocument('', '', nil);
  (DOMDoc as IDOMPersist).loadFromStream(Stream);
  memXML.Text := (DOMDoc as IDOMPersist).xml;
end;
```

When the stream isn't a proxy stream it works fine, and likewise, it also works correctly with a `TZDecompressionStream` proxy on Windows. However, it fails with an exception on OS X. The reason is that the `loadFromStream` call in the middle there tries to seek back on forth on OS X but not (by default) on Windows. The fix is easy enough however — save to a temporary memory stream first:

```
procedure TfrmXML.LoadXMLStream(Stream: TStream);
var
  DOMDoc: IDOMDocument;
  TempStream: TMemoryStream;
begin
  DOMDoc := GetDOM.createDocument('', '', nil);
  TempStream := TMemoryStream.Create;
  try
    TempStream.Size := Stream.Size - Stream.Position;
    Stream.ReadBuffer(TempStream.Memory^, TempStream.Size);
    (DOMDoc as IDOMPersist).loadFromStream(TempStream);
  finally
    TempStream.Free;
  end;
  memXML.Text := (DOMDoc as IDOMPersist).xml;
end;
```

A limited `Seek` implementation is not inherent to the idea of a proxy stream — it's just how `TZCompressionStream` and are `TZDecompressionStream` written.

Custom proxy streams: logging calls to a base stream

When implementing a custom `TStream` descendant, the methods of the base class that *must* be overridden are `Read`, `Write` and `Seek`. For a read-only stream, `Write` should either return 0, in which case `WriteBuffer` will raise an `EWriteError` exception, or raise an exception itself. If applicable, `SetSize` (a protected method) should be overridden too, since the default implementation does nothing. Lastly, `GetSize`, the getter method for the `Size` property, is also virtual. However, only override it if the size is directly reachable — the default implementation uses `Seek`, which provides a slightly roundabout way of finding the size.

The simplest sort of `TStream` descendant is a proxy stream that does not modify the data that is sent to or read from it. For example, a proxy class might be written to log `Read`, `Write` and `Seek` calls, aiding debugging. In such a case, the substance of the proxy's own `Read`, `Write` and `Seek` implementations would be to simply delegate to the base stream after recording what was requested.

In code terms, the definition of such a proxy class might look like the following. Here, the log is a `TStrings` instance, however it could be a `TTextWriter` or something else entirely:

```
type
  TProxyLogStream = class(TStream)
  strict private
    FBaseStream: TStream;
    FLog: TStrings;
    FOwnsStream: Boolean;
    procedure WriteToLog(const S: string; const Args: array of const);
  protected
    function GetSize: Int64; override;
    procedure SetSize(const NewSize: Int64); override;
  public
    constructor Create(ALog: TStrings; ABaseStream: TStream;
      AOwnsBaseStream: Boolean = False);
    destructor Destroy; override;
    function Read(var Buffer; Count: Integer): Integer; override;
    function Write(const Buffer; Count: Integer): Integer; override;
    function Seek(const Offset: Int64;
      Origin: TSeekOrigin): Int64; override;
    property BaseStream: TStream read FBaseStream;
    property Log: TStrings read FLog;
  end;
```

When overriding `SetSize` and `Seek`, you have a choice of 32 bit and 64 bit versions (in `TStream` they call each other, so you need to override one in each pair). Since overriding the 32 bit ones would potentially 'dumb down' 64 bit support in the base stream, we override the 64 bit versions here.

In implementing these methods, we will log the request, call the base stream implementation, and log the result:

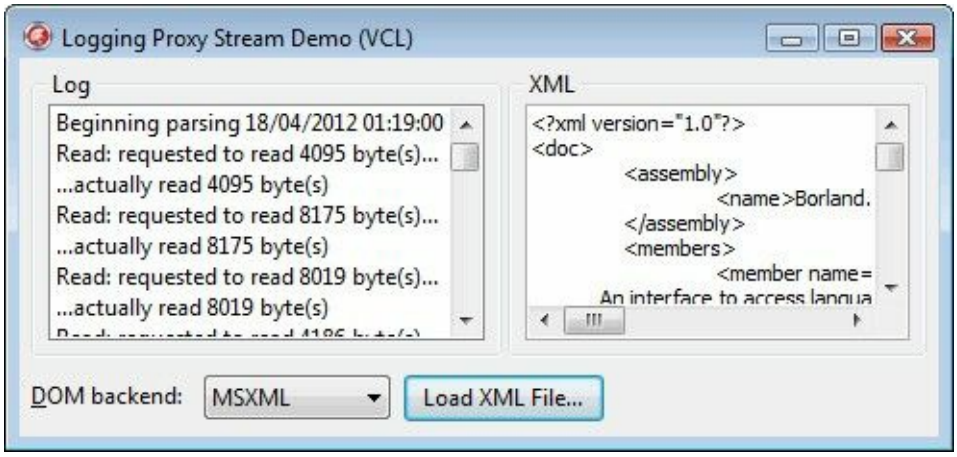
```
procedure TProxyLogStream.WriteToLog(const S: string;
  const Args: array of const);
begin
  FLog.Add(Format(S, Args))
end;

procedure TProxyLogStream.SetSize(const NewSize: Int64);
begin
  WriteToLog('SetSize: requested to set to %d byte(s)...', [NewSize]);
  BaseStream.Size := NewSize;
  WriteToLog('...completed without incident', []);
end;

function TProxyLogStream.Read(var Buffer; Count: Integer): Integer;
begin
  WriteToLog('Read: requested to read %d byte(s)...', [Count]);
  Result := BaseStream.Read(Buffer, Count);
  WriteToLog('...actually read %d byte(s)', [Result]);
end;

function TProxyLogStream.Write(const Buffer; Count: Integer): Integer;
begin
  WriteToLog('Write: requested to write %d byte(s)...', [Count]);
  Result := BaseStream.Write(Buffer, Count);
  WriteToLog('...actually wrote %d byte(s)', [Result]);
end;

function TProxyLogStream.Seek(const Offset: Int64;
  Origin: TSeekOrigin): Int64;
const
  SOrigin: array[TSeekOrigin] of string = (
```

```
    'beginning', 'current position', 'end');
begin
  if (Offset = 0) and (Origin = soCurrent) then
    WriteToLog('Seek: requested current position...', [])
  else
    WriteToLog('Seek: requested to seek %d byte(s) from %s...',
      [Offset, SOrigin[Origin]]);
  Result := BaseStream.Seek(Offset, Origin);
  WriteToLog('...returned %d', [Result]);
end;
```

The `GetSize` method just delegates to the base stream, the constructor initialises a few fields, and the destructor frees the base stream if the proxy has been set up to own it:

```
constructor TProxyLogStream.Create(ALog: TStrings;
  ABaseStream: TStream; AOwnsBaseStream: Boolean = False);
begin
  inherited Create;
  FBaseStream := ABaseStream;
  FOwnsStream := AOwnsBaseStream;
  FLog := ALog;
end;

destructor TProxyLogStream.Destroy;
begin
  if FOwnsStream then FBaseStream.Free;
  inherited Destroy;
end;

function TProxyLogStream.GetSize: Int64;
begin
  Result := BaseStream.Size;
end;
```

In the book's accompanying source code, a demo project for this class puts it to work logging what happens when an XML file is loaded using the RTL's two Document Object Model (DOM) backends (Delphi's XML support will be covered in the next chapter):



The substantive part of the demo wraps a `TFileStream` in a `TProxyLogStream` proxy before calling the DOM backend's `loadFromStream` method:

```
procedure TfrmLogProxyVCL.LoadXML(const AFileName: string);
var
  Document: IDOMPersist;
  Stream: TStream;
begin
  Document := GetDOM(cboDOMBackend.Text).createDocument(
    '', '', nil) as IDOMPersist;
  FLogStrings.Clear;
  try
    FLogStrings.Add('Beginning parsing ' + DateTimeToStr(Now));
    //wrap the base file stream in a logging wrapper
    Stream := TProxyLogStream.Create(FLogStrings,
      TFileStream.Create(AFileName, fmOpenRead), True);
    try
      Document.loadFromStream(Stream);
    finally
      Stream.Free;
    end;
    FLogStrings.Add('Ended parsing ' + DateTimeToStr(Now));
```

```
  finally
    lsbLog.Count := FLogStrings.Count;
  end;
  memXML.Text := Document.xml;
end;
```

The demo demonstrates how the loading behaviours of the two backends provided in the box (MSXML and ADOM) are quite different — where MSXML loads the file in reasonably sized chunks, presumably into a buffer it then scans through, ADOM will seek and read a few bytes at a time.

Custom proxy streams: buffering reads and writes

A slightly more complicated example of a proxy stream is one that adds generic buffering support. Here, the proxy will batch either read or write calls to the underlying stream, the point being to speed things up when only a few bytes are sent or requested at a time.

Similar to the `TZCompressionStream`/`TZDecompressionStream` case, it isn't really practical to define a single class to perform buffered reads *and* buffered writes, so a pair of classes, one for buffered reading and one for buffered writing, are in order. Nonetheless, not implementing proper seek functionality would be a bit lazy, so we won't do that — doing things properly just requires a bit of bookkeeping. Similarly, the fact buffering is performed in one direction shouldn't prevent non-buffered operations in the other.

First thing we need is a simple base class to prevent duplicating code unnecessarily:

```
type
  TBufferedStream = class abstract(TStream)
  strict private
    FBaseStream: TStream;
    FOwnsStream: Boolean;
  strict protected
    FBuffer: array of Byte;
    FBufferPos: Integer;
  public
    constructor Create(AStream: TStream;
      AOwnsStream: Boolean = False;
      ABufferSize: Integer = $10000); //64KB
    destructor Destroy; override;
    property BaseStream: TStream read FBaseStream;
    property OwnsStream: Boolean read FOwnsStream;
  end;

constructor TBufferedStream.Create(AStream: TStream;
  AOwnsStream: Boolean; ABufferSize: Integer);
begin
  inherited Create;
  FBaseStream := AStream;
  FOwnsStream := AOwnsStream;
  SetLength(FBuffer, ABufferSize);
end;

destructor TBufferedStream.Destroy;
begin
  if FOwnsStream then FBaseStream.Free;
  inherited Destroy;
end;
```

This sets up the base stream and initialises the buffer, which is defined as a dynamic array of bytes. A default buffer size of 64KB is used, however this can be customised as required.

The declaration of the buffered input stream class can be not much more than the necessary overrides:

```
type
  TBufferedInputStream = class(TBufferedStream)
  strict private
    FBufferSize: Integer;
    procedure EmptyBuffer;
  protected
    function GetSize: Int64; override;
    procedure SetSize(const NewSize: Int64); override;
  public
    destructor Destroy; override;
    function Read(var Buffer; Count: Integer): Integer; override;
    function Seek(const Offset: Int64;
      Origin: TSeekOrigin): Int64; override;
    function Write(const Buffer; Count: Integer): Integer; override;
```

```
  end;
```

In practical terms, buffering reads means that a minimum number of bytes will be requested from the base stream by the proxy, regardless of how much data the caller asks for; if that request is for less than the size of the buffer (or more exactly, for anything other than a multiple of the size of the buffer without remainder), then the remaining bytes are kept in memory by the proxy, to be passed on next time `Read` is called:

```
function TBufferedInputStream.Read(var Buffer;
  Count: Integer): Integer;
var
  BytesToReadOffBuffer: Integer;
  SeekPtr: PByte;
begin
  Result := 0;
  SeekPtr := @Buffer;
  while Count > 0 do
  begin
    if FBufferPos = FBufferSize then
    begin
      FBufferPos := 0;
      FBufferSize := BaseStream.Read(FBuffer[0], Length(FBuffer));
      if FBufferSize = 0 then Exit;
    end;
    BytesToReadOffBuffer := FBufferSize - FBufferPos;
    if BytesToReadOffBuffer > Count then
      BytesToReadOffBuffer := Count;
    Move(FBuffer[FBufferPos], SeekPtr^, BytesToReadOffBuffer);
    Inc(FBufferPos, BytesToReadOffBuffer);
    Inc(SeekPtr, BytesToReadOffBuffer);
    Dec(Count, BytesToReadOffBuffer);
    Inc(Result, BytesToReadOffBuffer);
  end;
end;
```

As appropriate, `Seek` calls must either be adjusted to take account of the buffer or cause the buffer itself to be reset:

```
function TBufferedInputStream.Seek(const Offset: Int64;
  Origin: TSeekOrigin): Int64;
begin
  if Origin = soCurrent then
    if (Offset >= 0) and (Offset <= (FBufferSize - FBufferPos)) then
    begin
      Inc(FBufferPos, Offset);
      Result := BaseStream.Seek(0, soCurrent) -
        FBufferSize + FBufferPos;
      Exit;
    end
    else
      Result := BaseStream.Seek(
        Offset - FBufferSize + FBufferPos, soCurrent)
  else
    Result := BaseStream.Seek(Offset, Origin);
  FBufferPos := 0;
  FBufferSize := 0;
end;
```

If the proxy does not own the base stream, and the buffer is not empty, then the base stream's seek position must be reset when the proxy is freed:

```
destructor TBufferedInputStream.Destroy;
begin
  if not OwnsStream then EmptyBuffer;
  inherited Destroy;
end;

procedure TBufferedInputStream.EmptyBuffer;
begin
  if (FBufferPos = 0) and (FBufferSize = 0) then Exit;
  BaseStream.Seek(FBufferPos - FBufferSize, soCurrent);
  FBufferPos := 0;
  FBufferSize := 0;
end;
```

The `Size` property getter can simply delegate to the base stream:

```
function TBufferedInputStream.GetSize: Int64;
begin
  Result := BaseStream.Size;
```

```
end;
```

Finally, `SetSize` and `Write` should either try and be clever and take account of the buffer, or just empty it and delegate to the base stream. We will do the latter:

```
procedure TBufferedInputStream.SetSize(const NewSize: Int64);
begin
  EmptyBuffer;
  BaseStream.Size := NewSize;
end;

function TBufferedInputStream.Write(const Buffer;
  Count: Integer): Integer;
begin
  EmptyBuffer;
  Result := BaseStream.Write(Buffer, Count);
end;
```

A buffered output class can follow a similar pattern:

```
type
  TBufferedOutputStream = class(TBufferedStream)
  protected
    function DoFlush: Boolean;
    function GetSize: Int64; override;
    procedure SetSize(const NewSize: Int64); override;
  public
    destructor Destroy; override;
    procedure Flush;
    function Read(var Buffer; Count: Integer): Integer; override;
    function Seek(const Offset: Int64;
      Origin: TSeekOrigin): Int64; override;
    function Write(const Buffer; Count: Integer): Integer; override;
  end;
```

Whereas the buffer of a buffered input stream is emptied at certain points, the buffer of a buffered output stream is 'flushed', i.e., written out to the base stream:

```
destructor TBufferedOutputStream.Destroy;
begin
  DoFlush;
  inherited Destroy;
end;

function TBufferedOutputStream.GetSize: Int64;
begin
  Flush;
  Result := BaseStream.Size;
end;

procedure TBufferedOutputStream.SetSize(const NewSize: Int64);
begin
  Flush;
  BaseStream.Size := NewSize;
end;

function TBufferedOutputStream.Read(var Buffer;
  Count: Integer): Integer;
begin
  Flush;
  Result := BaseStream.Read(Buffer, Count);
end;
```

In the case of `Seek`, we can avoid flushing if the method is called only to find the current position. In that case, we can return the current position of the base stream offset by the amount of data currently held in the buffer:

```
function TBufferedOutputStream.Seek(const Offset: Int64;
  Origin: TSeekOrigin): Int64;
begin
  if (Offset = 0) and (Origin = soCurrent) then
    Result := BaseStream.Seek(0, soCurrent) + FBufferPos
  else
  begin
    Flush;
    Result := BaseStream.Seek(Offset, Origin);
  end;
end;
```

Usually, operations that fail should raise an exception, however that is not always so in our case when it comes to flushing the buffer. One example is flushing in the destructor, since you should never raise an exception in a destructor. Another is in the `Write` implementation, since the difference between `Write` and `WriteBuffer` lies precisely in how `Write` does not raise an exception simply because not all the bytes requested could be written out.

The easiest way to implement this is to have a protected `DoFlush` method that returns `False` on failure, and a public `Flush` method that calls `DoFlush` and raises an exception if it returns `False`:

```
function TBufferedOutputStream.DoFlush: Boolean;
var
  BytesWritten: Integer;
begin
  while FBufferPos > 0 do
  begin
    BytesWritten := BaseStream.Write(FBuffer[0], FBufferPos);
    if BytesWritten = 0 then Exit(False);
    //move down the bytes still to be written
    Move(FBuffer[BytesWritten], FBuffer[0],
      FBufferPos - BytesWritten);
    //decrement the buffer write position
    Dec(FBufferPos, BytesWritten);
  end;
end;

procedure TBufferedOutputStream.Flush;
begin
  if not DoFlush then
    raise EWriteError.CreateRes(@SWriteError);
end;
```

`DoFlush` uses a looping construct to allow for the base stream only being able to write in chunks smaller than the buffer. While such behaviour is not common, it is worth taking account of (the implementation of `WriteBuffer` in `TStream` does, for example).

Lastly, `Write` both flushes the buffer when it is full and fills it in the first place:

```
function TBufferedOutputStream.Write(const Buffer;
  Count: Integer): Integer;
var
  BytesToBuffer: Integer;
begin
  Result := 0;
  while Count > 0 do
  begin
    if FBufferPos = Length(FBuffer) then
      if not DoFlush and (FBufferPos = Length(FBuffer)) then
        Exit;
    BytesToBuffer := Length(FBuffer) - FBufferPos;
    if BytesToBuffer > Count then BytesToBuffer := Count;
    Move(Buffer, FBuffer[FBufferPos], BytesToBuffer);
    Inc(FBufferPos, BytesToBuffer);
    Inc(Result, BytesToBuffer);
    Dec(Count, BytesToBuffer);
  end;
end;
```

In the book's accompanying source code, two small sample projects demonstrate `TBufferedOutputStream` and `TBufferedOutputStream`. In the first case, a dummy file 10 MB large is read in 2 byte chunks using a bare `TFileStream` and a file stream wrapped in a `TBufferedOutputStream` proxy. In the second, a bare `TFileStream` and a file stream wrapped in a `TBufferedOutputStream` proxy fight it out to be the quickest to write out a 4 MB file in 2 byte chunks. In both cases the proxy comes out much, much quicker, though be warned — in practice, reading or writing in tiny chunks is not very common.

9. ZIP, XML, Registry and INI file support

In this chapter, we will look at Delphi's 'in the box' support for some widely-used file formats — ZIP, XML and INI — together its support for manipulating the Windows Registry.

ZIP file support (TZipFile)

The ZIP format should need no introduction. A venerable way for end users to compress files, it is now a popular file structure in itself; for example, the DOCX and XLSX files produced by modern versions of Microsoft Office are really ZIP files with a custom extension.

In Delphi, the `System.Zip` unit provides a simple ZIP extractor and creator class, `TZipFile`. This has two basic styles of use. With the first, you call class methods to validate, create or extract a ZIP file in one line of code. With the second, in contrast, you instantiate the class in order to enumerate a ZIP, work with a ZIP entirely in memory, and so on.

One liners: using TZipFile's class methods

To test for whether a given file is a valid (well-formed) ZIP, use `TZipFile.IsValid`; to extract the contents of a ZIP file to a folder, call `TZipFile.ExtractZipFile`; to zip a folder's files, call `TZipFile.ZipDirectoryContents`:

```
ZipFile := 'C:\Users\CCR\Documents\Test.zip';
if TZipFile.IsValid(ZipFile) then
  TZipFile.ExtractZipFile(ZipFile, 'C:\DestDir')
else
  WriteLn(ZipFile, ' is not a valid ZIP file!');
//create a new ZIP file
TZipFile.ZipDirectoryContents('C:\FolderToZip', 'C:\ZippedFolder.zip');
```

These methods have the following characteristics:

- While `TZipFile.IsValid` takes a file name, it checks the actual file content, not the extension. This means passing it the path to (say) a new-style Microsoft Word document (`*.docx` or `*.docm`) should return `True` just as much as a non-corrupt `*.zip`.
- `TZipFile.ExtractZipFile` will respect any sub-directory structure involved in the source ZIP file, recreating it at the destination path. This mimics the '`Extract All...`' command in Windows Explorer.
- `TZipFile.ZipDirectoryContents` will replace any existing ZIP file rather than add files to it. Files will then be copied into the new archive recursively, recreating the sub-directory structure (if applicable) in the ZIP itself. This mimics the '`Send To|Compressed (zipped) Folder`' command in Windows Explorer.

Beyond the class methods

For more control than the class methods provide, you need to instantiate `TZipFile`. After invoking its constructor, call its `Open` method; this is passed either a file name or a stream (the stream must be managed independently), together a relevant value from the `TZipMode` enumeration:

```
TZipMode = (zmClosed, zmRead, zmReadWrite, zmWrite);
```

Use `zmWrite` to create a ZIP file afresh (in the case of the file name variant of `Open`, any existing file at the path specified will be deleted), `zmReadWrite` to edit an existing ZIP file, and `zmRead` to just read or enumerate an existing ZIP file's contents. At any time, you can check the `Mode` property to learn what mode the current file or stream was opened in, or (in the case of `zmClosed`) if there isn't a file or stream currently open at all.

Enumerating a ZIP file

When `Mode` is `zmRead` or `zmReadWrite`, the following properties can be read to get information about the ZIP file's contents:

```
//no. of files in the archive
property FileCount: Integer read GetFileCount;
//the name, comment or info structure for a specific entry
property FileName[Index: Integer]: string read GetFileName;
property FileComment[Index: Integer]: string read GetFileComment
  write SetFileComment;
property FileInfo[Index: Integer]: TZipHeader read GetFileInfo;
//get all names or info recs in one go
property FileNames: TArray<string> read GetFileNames;
property FileInfos: TArray<TZipHeader> read GetFileInfos;
//Comment relates to the whole ZIP, FileComment to a specific file
property Comment: string read FComment write FComment;
```

`FileInfo` and `FileInfos` return a fairly 'raw' data structure, `TZipHeader`, that contains all the saved information about a file:

```
TZipHeader = packed record
  MadeByVersion, RequiredVersion, Flag: UInt16;
  CompressionMethod: UInt16;
  ModifiedDateTime, CRC32, CompressedSize: UInt32;
```

```
  UncompressedSize: UInt32;
  FileNameLength, ExtraFieldLength: UInt16;
  FileCommentLength, DiskNumberStart: UInt16;
  InternalAttributes: UInt16;
  ExternalAttributes, LocalHeaderOffset: UInt32;
  FileName: RawByteString;
  ExtraField: TBytes;
  FileComment: RawByteString;
end;
```

Date/times in a ZIP file are encoded using the DOS format. On Windows the `FileDateToDateTime` function (`System.SysUtils`) will convert, otherwise you need to copy and paste from its Windows implementation so that you can call it on OS X too:

```
uses System.SysUtils;

function DOSFileDateToDateTime(FileDate: UInt32): TDateTime;
begin
  Result :=
    EncodeDate(
      LongRec(FileDate).Hi shr 9 + 1980,
      LongRec(FileDate).Hi shr 5 and 15,
      LongRec(FileDate).Hi and 31) +
    EncodeTime(
      LongRec(FileDate).Lo shr 11,
      LongRec(FileDate).Lo shr 5 and 63,
      LongRec(FileDate).Lo and 31 shl 1, 0);
end;
```

While ZIP date/times use the DOS format, file name paths use a forward slash. In fact, technically, a ZIP file has no sub-directory structure — it's just a flat list of items that by convention are interpreted to have paths if one or more forward slashes are in an item's name.

Here are the reading properties in action — `DOSFileDateToDateTime` is as above, and `TZipCompressionToString` is a small utility function provided by `System.Zip` itself:

```
function SizeStr(const ASize: UInt32): string;
begin
  if ASize < 1024 then
    FmtStr(Result, '%d bytes', [ASize])
  else
    FmtStr(Result, '%.0n KB', [ASize / 1024]);
end;

function ZipVerStr(const AVersion: UInt16): string;
begin
  FmtStr(Result, '%.1n', [AVersion / 10]);
end;

procedure EnumerateZipFile(ZipFile: TZipFile);
var
  I: Integer;
  Rec: TZipHeader;
begin
  WriteLn('ZIP file comment: ', ZipFile.Comment);
  for I := 0 to ZipFile.FileCount - 1 do
  begin
    WriteLn('"', ZipFile.FileName[I], '"');
    Rec := ZipFile.FileInfo[I];
    WriteLn('Compressed size: ',
      SizeStr(Rec.CompressedSize));
    WriteLn('Uncompressed size: ',
      SizeStr(Rec.UncompressedSize));
    WriteLn('Modified date/time: ', DateTimeToStr(
      DOSFileDateToDateTime(Rec.ModifiedDateTime)));
    WriteLn('Compression method: ', TZipCompressionToString(
      TZipCompression(Rec.CompressionMethod)));
    WriteLn('Comment: ', ZipFile.FileComment[I]);
    WriteLn('CRC: ', IntToHex(Rec.CRC32, 8));
    WriteLn('ZIP format version written as: ',
      ZipVerStr(Rec.MadeByVersion));
    WriteLn('Minimum ZIP version reader must understand: ',
      ZipVerStr(Rec.RequiredVersion));
    WriteLn;
  end;
end;
```

Extracting files and file data

To extract file data to an actual file, call the `Extract` method. This is overloaded to take either the source file name or its index, where `0` is the first archived file and `FileCount - 1` the last (in principle, a ZIP file can contain more than one item with the same name, even if this isn't a good idea given the way ZIP files tend to be used). Also available is `ExtractAll`:

```
procedure Extract(FileName: string; Path: string = '';
  CreateSubDirs: Boolean = True); overload;
procedure Extract(Index: Integer; Path: string = '';
  CreateSubDirs: Boolean = True); overload;
procedure ExtractAll(Path: string = '');
```

When you don't specify a path, the file is outputted to the current directory, so... specify a path! If you pass `False` for `CreateSubDirs`, the 'path' part of a ZIP'ed file name will be ignored.

Instead of extracting to a file, you might want to extract to an in-memory object. For example, if a ZIP contains a text file, you might want to extract its contents to a `TMemo`; similarly, you might want to extract a bitmap image file straight to a `TImage` control. This can be done via the `Read` method:

```
procedure Read(FileName: string; out Bytes: TBytes); overload;
procedure Read(Index: Integer; out Bytes: TBytes); overload;
procedure Read(FileName: string; out Stream: TStream;
  out LocalHeader: TZipHeader); overload;
procedure Read(Index: Integer; out Stream: TStream;
  out LocalHeader: TZipHeader); overload;
```

The stream versions here are rather lower-level than the `TBytes` ones — a stream object actually gets created for you, after which you need to take heed of the `LocalHeader` record instead of just reading from the stream naïvely. In general, the `TBytes` versions are therefore the ones to use, even if you want to have a stream output — just pass the returned array to a temporary `TBytesStream`.

For example, imagine a ZIP file contains two items, `Notes.txt` and `Picture.bmp`. The text file can be loaded into a `TMemo` control called `memNotes` like this:

```
var
  Bytes: TBytes;
  TempStream: TBytesStream;
begin
  ZipFile.Read('Notes.txt', Bytes);
  TempStream := TBytesStream.Create(Bytes);
  try
    memNotes.Lines.LoadFromStream(TempStream);
  finally
    TempStream.Free;
  end;
```

Similarly, the picture may be loaded into a FireMonkey `TImage` control called `imgPicture` like this:

```
var
  Bytes: TBytes;
  TempStream: TBytesStream;
begin
  ZipFile.Read('Picture.bmp', Bytes);
  TempStream := TBytesStream.Create(Bytes);
  try
    imgPicture.Bitmap.LoadFromStream(TempStream);
  finally
    TempStream.Free;
  end;
```

The equivalent code for a VCL `TImage` is almost the same:

```
var
  Bytes: TBytes;
  TempStream: TBytesStream;
begin
  ZipFile.Read('Picture.bmp', Bytes);
  TempStream := TBytesStream.Create(Bytes);
  try
    imgPicture.Picture.Bitmap.LoadFromStream(TempStream);
  finally
    TempStream.Free;
  end;
```

An alternative in either case would be to write out the data to a temporary file first, however using `Read` and `TBytesStream` will be the more efficient and performant approach, particularly if you have a lot of files to extract.

Creating and editing ZIP files

The converse of both `Extract` and `Read` is `Add`:

```
type
  TZipCompression = (zcStored, zcShrunk, zcReduce1, zcReduce2,
    zcReduce3, zcReduce4, zcImplode, zcTokenize, zcDeflate,
    zcDeflate64, zcPKImplode, zcBZIP2 = 12, zcLZMA = 14,
    zcTERSE = 18, zcLZ77, zcWavePack = 97, zcPPMdI1);

procedure Add(FileName: string; ArchiveFileName: string = '';
  Compression: TZipCompression = zcDeflate); overload;
procedure Add(Data: TBytes; ArchiveFileName: string;
  Compression: TZipCompression = zcDeflate); overload;
procedure Add(Data: TStream; ArchiveFileName: string;
  Compression: TZipCompression = zcDeflate); overload;
procedure Add(Data: TStream; LocalHeader: TZipHeader;
  CentralHeader: PZipHeader = nil); overload;
```

For the first variant, not passing something for `ArchiveFileName` means the source `FileName` minus its path is used. To specify a 'sub-directory', pass a string for `ArchiveFileName` with embedded forward slashes:

```
ZipFile.Add(SourceFile, 'Child dir/' + ExtractFileName(SourceFile));
```

The same thing needs to be done if a directory structure is desired when using any of the other `Add` variants too:

```
ZipFile.Add(SourceBytes, 'Child dir/My file.dat');
```

By default, the now widely-understood UTF-8 extension to the original ZIP format will be used, unless you are editing an existing ZIP file that hasn't been written with it enabled. Using the UTF-8 extension means file names can contain non-ASCII characters. If you wish to force ASCII file names only, set the `UTF8Support` property to `False` *before* adding any files. (Note this setting is only about file names — there are no restrictions on file contents.)

A moment ago we saw how to extract a ZIP'ed text file or bitmap straight into a `TMemo` or `TImage`. Here's how to go the other way, ZIP'ing the contents of a `TMemo` or `TImage` without creating separate files on disk first:

```
var
  Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    //TMemo called Memo1, UTF-8 with BOM
    Memo1.Lines.SaveToStream(Stream, TEncoding.UTF8);
    Stream.Position := 0;
    ZipFile.Add(Stream, 'Memo1.txt');
    //TMemo called Memo2, UTF-8 without BOM
    ZipFile.Add(TEncoding.UTF8.GetBytes(Memo2.Text), 'Memo2.txt');
    //FireMonkey Timage called imgPicture
    Stream.Clear;
    imgPicture.Bitmap.SaveToStream(Stream);
    Stream.Position := 0;
    ZipFile.Add(Stream, 'Picture.png');
    //VCL Timage called imgPicture
    Stream.Clear;
    imgPicture.Picture.Bitmap.SaveToStream(Stream);
    Stream.Position := 0;
    ZipFile.Add(Stream, 'Picture.bmp');
  finally
    Stream.Free;
  end;
```

The writing process will be completed when you either free the `TZipFile` object, open another zip file or stream, or call the `Close` method.

Extending TZipFile

The ZIP standard allows for a variety of different compression methods. Unfortunately, while `TZipFile` supports the most common method ('deflate'), along with the option not to compress at all ('stored'), it doesn't support less common ones. Moreover, even for deflate, it doesn't allow specifying the compression sub-type ('fastest', 'normal' or 'maximum').

Nonetheless, `TZipFile` is extensible to the extent you can plug in support for other compression methods, if you have the code for them. Doing this involves calling the `RegisterCompressionHandler` class method, usually in the `initialization` section of a unit, to which you pass the compression type being implemented, together with a pair of anonymous methods. These construct a new instance of `TStream` descendant to read and write from, given a raw input steam:

```
type
  TStreamConstructor = reference to function(InStream: TStream;
    const ZipFile: TZipFile; const Item: TZipHeader): TStream;

class procedure RegisterCompressionHandler(
  CompressionMethod: TZipCompression;
  CompressStream, DecompressStream: TStreamConstructor);
```

In the case of a compression method that already has a handler registered, `RegisterCompressionHandler` will simply replace the old handler with the new.

Demonstrating how to actually implement one of the more exotic compression methods is outside the scope of this book. However, showing how to customise the compression level used by the stock 'deflate' implementation is easy enough. This is because `TZipFile` just delegates to the `TZCompressionStream` and `TZDecompressionStream` classes we met in the previous chapter. All we need do, then, is delegate similarly:

```
unit EnforceMaxZipFileCompression;

interface

implementation

uses
  System.Classes, System.ZLib, System.Zip;

initialization
  TZipFile.RegisterCompressionHandler(zcDeflate,
    function(InStream: TStream; const ZipFile: TZipFile;
      const Item: TZipHeader): TStream
    begin
      Result := TZCompressionStream.Create(clMax, InStream);
    end,
    function(InStream: TStream; const ZipFile: TZipFile;
      const Item: TZipHeader): TStream
    begin
      Result := TZDecompressionStream.Create(InStream);
    end);
end.
```

Add this unit to a `uses` clause, and `TZipFile` will now save out files with a preference for smaller file sizes over speed of operation.

XML support

XML — eXtensible Markup Language — is a kind of file format (or meta-format) that is hard to avoid. Delphi's support for it is quite extensive, however the coverage varies between both product edition and target platform. As a result, this book will only provide a tour of those features with the broadest support.

TXMLDocument

At the centre of Delphi's XML functionality is the `TXMLDocument` class. This allows working with XML documents with an interface akin to the standard Document Object Model (DOM), though at a slightly higher level — internally, `TXMLDocument` in fact delegates to an appropriate DOM backend to do the hard work.

`TXMLDocument` is implemented as both a component and an object you can use via a corresponding interface type (`IXMLDocument`). Being a component means you can drop it onto a form or data module at design time. Do so, and the following properties will be shown in the Object Inspector:

- `DOMVendor` specifies the parsing backend `TXMLComponent` should use. Included in XE2 are two, MSXML (Microsoft's COM-based XML parser) and ADOM ('Alternative Document Object Model'). MSXML is the default on Windows and only available on that platform; ADOM (an open source project by Dieter Köhler, written in Delphi) is the default when targeting the Mac, and available on Windows too.
- `FileName` is a `string` property that specifies an XML file on disk to load into the component.
- Alternatively, `XML` is a `TStrings` property in which you can input the source document's content directly. If both `XML` and `FileName` are set, then `XML` takes precedence.
- `Active` is a `Boolean` property; set it to `True` to load the XML structure specified by either `FileName` or `XML` into memory. An exception will be raised when doing that if the source XML couldn't be parsed for some reason.
- `Options` is a set property controlling how the `TXMLDocument` behaves once data has been successfully loaded. Its possible elements are `doNodeAutoCreate`, `doNodeAutoIndent`, `doAttrNull`, `doAutoPrefix`, `doNamespaceDecl` and `doAutoSave`, with all but `doNodeAutoIndent` and `doAutoSave` enabled by default.
- If you add `doNodeAutoIndent` to `Options`, adding a new node to the source structure inserts appropriate indentation when the revised XML is read back out again. The `NodeIndentStr` property then determines what sort of indentation is actually inserted: a tab, one space character, two space characters, etc.
- If you add `doAutoSave` to `Options`, setting `Active` to `False` when it was `True` before causes either the `XML` property or the file pointed to by `FileName` to be updated with the new data.
- `ParseOptions` is another set property, this time controlling how the source XML is loaded in the first place. Its possible elements are `poResolveExternals`, `poValidateOnParse`, `poPreserveWhiteSpace`, and `poAsyncLoad`; any changes must be made prior to `Active` being set to `True`.

Of the `ParseOptions` elements, only `poPreserveWhiteSpace` is supported by the ADOM backend, with the rest being MSXML specific. When `poPreserveWhiteSpace` is included, technically superfluous whitespace in the source XML (e.g. space or tab characters) should survive editing cycles.

Of the other `ParseOptions` elements, `poResolveExternals` determines whether any external includes and imports in the source XML are resolved when `Active` is set to `True`, `poValidateOnParse` causes the source XML to be validated on `Active` being set to `True`, and `poAsyncLoad` causes the XML structure to be loaded in a separate thread. Keeping `poAsyncLoad` excluded (which is the default) means you can assume the structure is loaded as soon as the `Active` property setter returns.

Beyond the properties just listed, `TXMLComponent` also has various events assignable at design-time: `AfterClose`, `AfterNodeChange`, `AfterOpen`, `BeforeClose`, `BeforeNodeChange`, `BeforeOpen` and `OnAsyncLoad`. (Notice the missing 'On-' prefix for most of them!) Here, `OnAsyncLoad` is used with `poAsyncLoad`. Assign it, and your handler will be called at various points in the loading process, the `AsyncLoadState` parameter having the value of 4 once loading is complete:

```
procedure TForm1.XMLDocument1AsyncLoad(Sender: TObject;
  AsyncLoadState: Integer);
begin
  if AsyncLoadState = 4 then
  begin
    //work with loaded XML...
  end;
end;
```

Using TXMLDocument at runtime

In practice, you don't have to create `TXMLDocument` at design-time. In fact, for cross platform code it is better not to — in the IDE, the `DOMProvider` property gets initialised to MSXML, which can quickly cause compilation errors if you add OS X as a target platform. If you really must use `TXMLDocument` at design-time and want to target OS X, set `DOMProvider` to `ADOM - v4`; if `Xml.Win.msxmldom` has already been added to the form or data module's `uses` clause, delete it.

When explicitly instantiating `TXMLDocument` at runtime, best practice is to use one of the `NewXMLDocument`, `LoadXMLData` or `LoadXMLDocument` helper functions instead of `TXMLDocument.Create` directly. All three return an `IXMLDocument` interface reference:

```
uses Xml.XmlIntf, Xml.XmlDoc;

const
  XMLString =
    '<?xml version="1.0"?> ' +
    '<philosophers> ' +
    ' <philosopher name="Kant"> ' +
      ' <idea>categorical imperative</idea> ' +
      ' <idea>formal idealism</idea> ' +
    ' </philosopher> ' +
    ' <philosopher name="Hegel"> ' +
      ' <idea>dialectic</idea> ' +
      ' <idea>historicism</idea> ' +
    ' </philosopher> ' +
    '</philosophers>';
var
  Doc: IXMLDocument;
begin
  //load from a file
  Doc := LoadXMLDocument('C:\Users\CCR\Documents\Some file.xml');
  //load from a string
  Doc := LoadXMLData(XMLString);
  //create a new XML document afresh
  Doc := NewXMLDocument;
```

When `TXMLDocument` is used through an `IXMLDocument` interface reference, it is properly reference counted and so should *only* be accessed through the interface and never a variable typed to `TXMLDocument` itself.

Technically, what controls reference counting being used or not is whether `nil` is passed to `Create` for the component's owner. Thus, if you avoid either of the three helper functions just illustrated yet nevertheless still explicitly construct a `TXMLDocument` at runtime, whether you access it as a class or an interface reference should depend on whether you gave it an owner or not:

```
var
  DocComp: TXMLDocument;
  DocIntf: IXMLDocument;
begin
  DocComp := TXMLDocument.Create(Self); //correct
  DocIntf := TXMLDocument.Create(nil);  //also correct
  DocComp := TXMLDocument.Create(nil);  //wrong!
  DocIntf := TXMLDocument.Create(Self); //also wrong!
```

Enumerating a TXMLDocument

Once data has been loaded with the `Active` property set to `True`, the `ChildNodes` property of `TXMLDocument`/`IXMLDocument` returns a list of the XML structure's top level nodes. These will number the so-called 'document element' or root data node, along with any top level comment nodes and processing instructions (the root data node can also be accessed via the `DocumentElement` property). For example, the following XML document will contain three top level nodes: the XML version node (a 'processing instruction'), the comment, and the document element ('books'):

```
<?xml version="1.0"?>
<!-- This is a comment -->
<books>
  <book title="Philosophy of Right" author="GWF Hegel"/>
  <book>
    <title>Critique of Practical Reason</title>
    <author>Immanuel Kant</author>
    <year>1788</year>
  </book>
</books>
```

Lists of nodes are represented by the `IXMLNodeList` interface. This type has a `Count` property and a default `Nodes` property,

together with `Add`, `Insert`, `Delete`, `Remove`, `FindNode`, `IndexOf`, `BeginUpdate`, `EndUpdate`, `First` and `Last` methods. Individual nodes can be accessed either by index or by name, assuming they have a name:

```
var
  S: string;
begin
  S := XMLDocument1.ChildNodes[2].NodeName; //'books'
  XMLDocument1.ChildNodes.Delete('books');
  XMLDocument1.ChildNodes.Delete(0);
```

Items themselves, along with the `DocumentElement` property of `IXMLDocument`, are instances of the `IXMLNode` interface. This has numerous members. For traversing nodes, it has `NextSibling` and `PreviousSibling` methods — both return `nil` if there is no sibling reference to return. Beyond them, it also has the following properties:

- `NodeName` returns or sets the node's name, which will be fully-qualified if XML namespaces are used.
- `LocalName` also returns or sets the node's name, but without any namespace identifier if applicable.
- `AttributeNodes` returns an `IXMLNodeList` of the node's attributes — in the case of the first 'book' node in the XML structure above, that will be for 'title' and 'author'; in the case of the second, none at all. As a small shortcut, there is also an `Attributes` array property to get or set an attribute's value directly by name.
- `ChildNodes` returns another `IXMLNodeList`, this time for any nodes (excluding attributes) that the node parents. In the case of the second 'book' node above, that would mean returning a list of three children, 'title', 'author' and 'year'. Similar to `Attributes`, a `ChildValues` array property exists too to get or set a child node's value directly by name or index.
- `Text` returns or sets the node's value as a string. In the case of an 'element' node that has child elements (such as the 'books' node above), reading this property will raise an exception.
- `NodeValue` is like `Text`, only typed to a variant rather than a string. An empty value will return `Null`, and like `Text`, attempting to get the value for an element node with child elements will raise an exception.
- `NodeType` returns an enumerated value that specifies what sort of node it is:

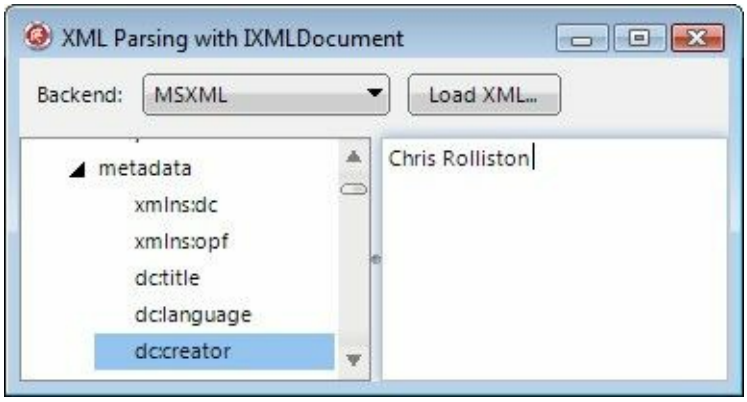
```
TNodeType = (ntReserved, ntElement, ntAttribute, ntText, ntCData,
  ntEntityRef, ntEntity, ntProcessingInstr, ntComment,
  ntDocument, ntDocType, ntDocFragment, ntNotation);
```

As an example of enumerating an `IXMLDocument` using these properties, here's how you might go about loading all attribute and element nodes into a FireMonkey `TTreeView` control:

```
type
  TTreeViewNodeItem = class(TTreeViewItem)
    Node: IXMLNode;
  end;

procedure LoadXMLNodeStructure(const ADocument: IXMLDocument;
  ATreeView: TTreeView);

  procedure AddNode(AParent: TControl; const ANode: IXMLNode);
  var
    I: Integer;
    NewItem: TTreeViewNodeItem;
  begin
    if (ANode = nil) or not (ANode.NodeType in [ntElement,
      ntAttribute]) then Exit;
    NewItem := TTreeViewNodeItem.Create(ATreeView);
    NewItem.Node := ANode;
    NewItem.Text := ANode.NodeName;
    AParent.AddObject(NewItem);
    for I := 0 to ANode.AttributeNodes.Count - 1 do
      AddNode(NewItem, ANode.AttributeNodes[I]);
    if ANode.HasChildNodes then
      for I := 0 to ANode.ChildNodes.Count - 1 do
        AddNode(NewItem, ANode.ChildNodes[I]);
  end;
begin
  ATreeView.BeginUpdate;
  try
    ATreeView.Clear;
    AddNode(ATreeView, ADocument.DocumentElement);
  finally
    ATreeView.EndUpdate;
  end;
```

```
end;
```

Amongst this book's sample projects is a simple XML browser:



Naïvely, you might think the tree view's `OnChange` event could be handled by simply assigning the select node's `Text` property to the memo. However, given the potentially exception-raising behaviour of the `Text` property just mentioned, the handler needs to be a bit more precise in what it does:

```
procedure TXMLForm.StructureTreeChange(Sender: TObject);
var
  Node: IXMLNode;
begin
  if StructureTree.Selected = nil then Exit;
  Node := (StructureTree.Selected as TTreeViewNodeItem).Node;
  if (Node.NodeType = ntElement) and
     (Node.ChildNodes.Count > 0) and
     (Node.ChildNodes[0].NodeType <> ntText) then
    memContent.Text := ''
  else
    memContent.Text := Node.Text;
end;
```

The tricky part here is how the text part of an element node is itself a node with the `ntText` type!

Creating a new XML document dynamically

When creating a new XML document afresh, the root node needs to be set first by assigning the `DocumentElement` property. The node for this can be created using the `CreateNode` method of the controlling `IXMLDocument`. Once done, you can add child nodes either by calling `CreateNode` again, or the `AddChild` method of the parent node:

```
var
  Doc: IXMLDocument;
  XML: string;
begin
  Doc := NewXMLDocument;
  Doc.Options := Doc.Options + [doNodeAutoIndent];
  Doc.DocumentElement := Doc.CreateNode('myroot');
  Doc.DocumentElement.Text := 'Hello world';
  //add by explicitly creating a child node
  Child := Doc.CreateNode('FirstChild');
  Child.Text := 'Hello again';
  Doc.DocumentElement.ChildNodes.Add(Child);
  //add in a slightly more succint fashion
  Doc.DocumentElement.AddChild('SecondChild').Text :=
    'Hello yet again';
  //save the new XML
  Doc.SaveToXML(XML);
  ShowMessage(XML);
```

This produces the following output:

```
<?xml version="1.0"?>
<myroot>Hello world  <FirstChild>Hello again</FirstChild>
  <SecondChild>Hello yet again</SecondChild>
</myroot>
```

As previously noted, the `Options` property of `TXMLDocument` and `IXMLDocument` includes `doNodeAutoCreate` by default. This has the affect of automatically creating a node if it doesn't already exist, given the name passed to the `ChildNodes` default property:

```
Doc.DocumentElement.ChildNodes['ThirdChild'].Text :=
  'Hello yet again'; //never raises error with doNodeAutoCreate
```

This can be convenient, particularly when setting attributes:

```
Doc.DocumentElement.ChildNodes['SecondChild'].Attributes[
  'MyAttr'] := 'Blah blah blah';
```

However, its downside is that typos won't get picked up.

Once edited you your satisfaction, an `IXMLDocument` instance can be saved to a string using either the `SaveToXML` method as above, or a file using `SaveToFile` or a stream using `SaveToStream`. Alternatively, if the either the `FileName` or `XML` properties had been initialised and `Active` set to `True`, ensure `doAutoSave` is in `Options` and set `Active` to `False`.

Using the DOM parsers directly

Internally, `TXMLDocument` uses a Document Object Model (DOM) parser to do the grunt work. The DOM is a standardised interface for manipulating XML in memory; in Delphi, it is represented by a set of `IDOMxxx` interfaces declared in the `Xml.XmlDom` unit. While `TXMLDocument` works OK in general, using those lower-level interfaces directly will be more efficient (`TXMLDocument` can start to drag on very large documents).

Another reason to use the `IDOMxxx` interfaces directly is to find nodes using 'XPath' expressions. In this situation you can still use `TXMLDocument`/`IXMLDocument` since the underlying `IDOMxxx` interfaces are exposed by the `IXMLxxx` ones: specifically, `IXMLDocument` has a `DOMDocument` property to get or set the underlying `IDOMDocument`, and `IXMLNode` a readable `DOMNode` property that exposes the underlying `IDOMNode`. From `DOMNode`, query for the `IDOMNodeSelect` interface to be able to send XPath expressions (more on which shortly).

Alternatively, you can use the `IDOMxxx` interfaces completely independently of `TXMLDocument`. To do this, you must first must call the `GetDOM` global function of `Xml.XmlDom` to retrieve an `IDOMImplementation` instance. This function takes a single string parameter to specify which DOM parser to use; don't pass anything if you don't care, in which case the platform default will be used (MSXML on Windows, ADOM on OS X).

You can get a list of installed backends via the `DOMVendors` global variable, however without any input on your part this will only contain the entry for the platform default. On Windows ADOM will be added to if you add Xml.AdomXmlDom to a uses clause somewhere.

Once you have an `IDOMImplementation`, call the `createDocument` method to get an `IDOMDocument`:

```
uses Xml.XmlDom;

var
  Document: IDOMDocument;
begin
  Document := GetDOM.createDocument('', '', nil);
```

In itself, `IDOMDocument` provides no methods to read and write whole XML documents. Nevertheless, they are available once you query for `IDOMPersist`:

```
const
  XMLString =
    '<?xml version="1.0"?> ' +
    '<philosophers> ' +
    ' <philosopher name="Schopenhauer"> ' +
      ' <idea>pessimism</idea> ' +
      ' <idea>will</idea> ' +
    ' </philosopher> ' +
    ' <philosopher name="Nietzsche"> ' +
      ' <idea>perspectivism</idea> ' +
      ' <idea>amor fati</idea> ' +
    ' </philosopher> ' +
    '</philosophers>';
var
  Doc: IDOMDocument;
  Success: Boolean;
begin
  Doc := GetDOM.createDocument('', '', nil);
  //load from a file
  Success := (Doc as IDOMPersist).load('C:\Misc\Stuff.xml');
  //load from a string
  Success := (Doc as IDOMPersist).loadxml(XMLString);
  //load from a stream
  Success := (Doc as IDOMPersist).loadFromStream(MyStream);
```

Each of `load`, `loadxml` and `loadFromStream` return a `Boolean` result indicating whether the XML document was loaded successfully. When they return `False`, details of the error can be found by querying for `IDOMParseError`:

```
var
  Doc: IDOMDocument;
  Error: IDOMParseError;
begin
  Doc := GetDOM.createDocument('', '', nil)
  if not (Doc as IDOMPersist).loadxml(XMLString) then
  begin
    Error := (Doc as IDOMParseError);
    WriteLn('Error code: ', Error.errorCode);
    WriteLn('Reason: ', Error.reason);
    WriteLn('Source text: ', Error.srcText);
    WriteLn('Line: ', Error.line);
    WriteLn('Line position: ', Error.linePos);
    WriteLn('File position: ', Error.filePos);
    //...
```

Prior to calling one of the load methods, you can set the same options exposed by the `ParseOptions` property of `TXMLDocument` by querying for the `IDOMParseOptions` interface:

```
procedure SetParseOptions(const ADocument: IDOMDocument);
var
  OptionsIntf: IDOMParseOptions;
begin
  OptionsIntf := ADocument as IDOMParseOptions;
  OptionsIntf.PreserveWhiteSpace := True;
  //the following remain MSXML specific!
  OptionsIntf.ResolveExternals := True;
  OptionsIntf.Validate := True;
  OptionsIntf.Async := False; //this is the default
end;
```

Once loaded, the DOM structure can be enumerated in a similar way to `IXMLDocument`. Here's the tree view code given previously, now rewritten to use `IDOMxxx` instead of `IXMLxxx`:

```
type
  TTreeViewNodeItem = class(TTreeViewItem)
    Node: IDOMNode;
  end;

procedure LoadXMLNodeStructure(const ADocument: IDOMDocument;
  ATreeView: TTreeView);
  procedure AddNode(AParent: TControl; const ANode: IXMLNode);
  var
    I: Integer;
    NewItem: TTreeViewNodeItem;
  begin
    if ANode = nil then Exit;
    case ANode.NodeType of
      ELEMENT_NODE, ATTRIBUTE_NODE: { OK }
    else Exit;
    end;
    NewItem := TTreeViewNodeItem.Create(ATreeView);
    NewItem.Node := ANode;
    NewItem.Text := ANode.NodeName;
    AParent.AddObject(NewItem);
    if ANode.Attributes <> nil then
      for I := 0 to ANode.Attributes.Length - 1 do
        AddNode(NewItem, ANode.Attributes[I]);
    if ANode.HasChildNodes then
      for I := 0 to ANode.ChildNodes.Length - 1 do
        AddNode(NewItem, ANode.ChildNodes[I]);
  end;
begin
  ATreeView.BeginUpdate;
  try
    ATreeView.Clear;
    AddNode(ATreeView, ADocument.DocumentElement);
  finally
    ATreeView.EndUpdate;
  end;
end;
```

Unlike with `IXMLxxx`, there is no auto-creation of nodes when using the `IDOMxxx` interfaces. Furthermore, various helper methods are missing, the `Count` method becomes `Length`, and some quirks are apparently added: in particular, the `Attributes` property of `IDOMNode` will be `nil` if there are no attributes defined, and in the case of an element node with text, the text will *only* appear as an explicit child 'text' node. For example, consider the following XML snippet:

```
<MyNotes>This is some text</MyNotes>
```

If `Node` is an `IDOMNode` representing `MyNotes`, then `Node.Attributes` will return `nil`, `Node.ChildNodes.Length` will return `1`, and `Node.ChildNodes[0].NodeValue` will return `This is some text`. Conversely, adding an element node with text means adding *two* nodes:

```
function AddElementWithText(const ADoc: IDOMDocument;
  const AParent: IDOMNode; const AName, AText: string): IDOMElement;
begin
  Result := ADoc.CreateElement(AName);
  Result.AppendChild(ADoc.CreateTextNode(AText));
  ADoc.AppendChild(Result)
end;
```

In order to save XML back out of an `IDOMDocument`, query once more for `IDOMPersist`, then call either `Save` to save to a file, `SaveToStream` to save to a stream, or read the `Xml` property to get the XML as a string:

```
procedure OutputXML(const ADocument: IDOMDocument);
var
  XML: string;
begin
  ShowMessage('XML to be outputted:' + SLineBreak +
    (ADocument as IDOMPersist).XML);
  (ADocument as IDOMPersist).Save('C:\Data\Some file.xml');
end;
```

Finding nodes using XPath expressions

Using 'XPath' expressions, you can locate nodes in an XML structure using strings made up of node paths and simple filtering expressions. The XPath syntax is standardised, which means the expression format you use in Delphi is the same as the one a JavaScript programmer (for example) would use.

Since XPath support is not surfaced by the `IXMLxxx` interfaces, you *must* drop down to the `IDOMxxx` level. On the other hand, both the MSXML and ADOM backends implement it, so you can employ exactly the same code regardless of which one you use.

In practice, the system works by querying for the `IDOMNodeSelect` interface from an `IDOMNode` or `IDOMDocument` instance. `IDOMNodeSelect` is composed of two methods: `SelectNode`, which returns the first matched `IDOMNode` or `nil` if nothing was matched, and `SelectNodes`, which returns an `IDOMNodeList` instance (whether `nil` is returned on failure rather than an empty list will depend on the backend, so you should check for both). Both these methods take a single string parameter that specifies the node or nodes to look for.

At its simplest, this parameter might be just a straightforward path. In such a case, use the forward slash character to delimit element nodes in the node hierarchy. For example, consider the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<products>
  <product InRADStudioFamily="true">
    <name>Delphi</name>
    <currentVersion>XE2</currentVersion>
    <editions>
      <edition name="Starter" NewUserPrice="142"
        UpgradePrice="106" />
      <edition name="Professional"
        NewUserPrice="642" UpgradePrice="356" />
      <edition name="Enterprise" NewUserPrice="1428"
        UpgradePrice="928" />
      <edition name="Architect" NewUserPrice="2499"
        UpgradePrice="1642" />
    </editions>
  </product>
  <product InRADStudioFamily="true">
    <name>C++Builder</name>
    <currentVersion>XE2</currentVersion>
    <editions>
      <edition name="Starter" NewUserPrice="142"
        UpgradePrice="106" />
      <edition name="Professional" NewUserPrice="642"
        UpgradePrice="356" />
      <edition name="Enterprise" NewUserPrice="1428"
        UpgradePrice="928" />
      <edition name="Architect" NewUserPrice="2499"
        UpgradePrice="1642" />
    </editions>
```

```
  </product>
  <product InRADStudioFamily="false">
    <name>JBuilder</name>
    <currentVersion>2008 R2</currentVersion>
    <editions>
      <edition name="Professional" NewUserPrice="356"
        UpgradePrice="179" />
      <edition name="Enterprise" NewUserPrice="1071"
        UpgradePrice="536" />
    </editions>
  </product>
</products>
```

Here, the document element is called 'products'. This then has a series of 'product' child nodes, each of which have an attribute called 'InRADStudioFamily' and a set of further nodes: 'name', 'currentVersion' and 'editions'. The last of these involves yet more sub-nodes, however each 'edition' node is composed of attributes only.

Given all that, and assuming the XML has been loaded into an `IDOMDocument`, the following code returns a list containing each of the product name nodes:

```
procedure Foo(const Doc: IDOMDocument);
var
  Nodes: IDOMNodeList;
begin
  Nodes := (Doc as IDOMNodeSelect).SelectNodes(
    '/products/product/name');
```

A path that begins with a forward slash is an absolute path; one without, a path that is relative to the node you are calling `SelectNodes` on. However, in this particular example the distinction is moot, given we are calling `SelectNodes` from the document itself.

Since we are working at the `IDOMxxx` level, the nodes returned by a path such as `'/products/product/name'` will be element nodes that you have to look into to retrieve the actual text (i.e., `'Delphi'`, `'C++Builider'` and `'JBuilder'`):

```
function GetTextNode(const Element: IDOMNode): IDOMNode);
var
  I: Integer;
begin
  for I := Element.ChildNodes.Length - 1 downto 0 do
  begin
    Result := Element.ChildNodes[I];
    if Result.NodeType = TEXT_NODE then Exit;
  end;
  Result := nil;
end;

//...
var
  I: Integer;
  S: string;
begin
  S := 'Found values: ';
  for I := 0 to Nodes.Length - 1 do
    S := S + GetTextNode(Nodes[I]).NodeValue;
  ShowMessage(S);
```

An alternative is to append `'/text()'` to the search path. This has the effect of returning the text nodes directly: `'/products/product/name/text()'`.

For a full account of the XPath syntax, you should refer to a good XPath reference (a nice place to start is the W3Schools website: `http://www.w3schools.com/XPath/`). Here are some highlights:

- To search for an attribute, do the same as for an element node, only prepending the attribute name with an @ sign: `'/products/product/@InRADStudioFamily'`.
- Node collections, such as 'products' and the series of 'editions' nodes in our example, can be indexed. Since XPath indexing begins at 1, `'/products/product[2]/name/text()'` will therefore return a single text node with the value `'C++Builder'`.
- To retrieve the last node in a collection, use the `last()` pseudo-function: `'/products/product[last()]/name/text()'` returns a text node with the value `'JBuilder'`. To retrieve the next-from-last node, use `last()-1`, the third-from last `last()-2`, and so on. If you take away too many items, then nothing will be returned.
- Using two forward slashes means 'find such a node (or attribute) *anywhere* in the XML structure'. Thus,

`'//@InRADStudioFamily'` will return three attribute nodes with the values true, true and false.

- Returned nodes can be filtered by embedding simple Boolean expressions, each expression being wrapped in square brackets. For example, `'//product[@InRADStudioFamily="true"]/name'` will return the element nodes for Delphi and C++Builder, but not JBuilder. Be careful to get the casing right, since XML is case sensitive, unlike Delphi. (Another XPath/Delphi difference is that string literals in the former can be delimited with either single or double quotes, as indicated in the previous example.)
- Supported operators for filtering include those for equality (`=`), not equal (`!=`), less than (`<`), less than or equal (`<=`), greater than (`>`), greater than or equal (`>=`), logical `or`, logical `and`, addition (`+`), subtraction (`-`), multiplication (`*`), division (`div`) and modulus (`mod`). Notice these match the Delphi operators, with the major exception of the 'not equal' operator which is `!=` in XPath but `<>` in Delphi.
- Both `or` and `and` work like in Delphi, so that

```
'/products/product[name="Delphi" or name="JBuilder"]/currentVersion'
```

  returns nodes for XE2 and 2008 R2, but

```
'/products/product[name="Delphi" and name="JBuilder"]/currentVersion'
```

  returns nothing, since no product node has a child node called *both* Delphi and JBuilder.

- The | symbol can be used to combine two different result sets. An alternative way to find the nodes for Delphi and JBuilder's current versions is therefore

```
'/products/product[name="Delphi"]/currentVersion | ' +
'/products/product[name="JBuilder"]/currentVersion'
```

- Filtering can be done in multiple places along the search path. For example,

```
'/products/product[name='Delphi']/editions/' +
'edition[@NewUserPrice>2000]/@name'
```

  will return the names of all Delphi product editions that have a new user price greater than £2000.

Windows Registry access

The Delphi RTL provides two classes for working with the Windows Registry, the lower level `TRegistry` and the higher level `TRegistryIniFile`, both of which are declared in the `System.Win.Registry` unit. If you just wish to read and write settings of your own design, `TRegistryIniFile` (which we will be looking at later on the chapter) is generally the better option. This is because `TRegistry` requires knowledge of how the Registry works to be used effectively where `TRegistryIniFile` doesn't. Nonetheless, if you wish to read or write system-defined keys, `TRegistry` comes into its own.

Background

The Registry is organised into hierarchical trees of 'keys'. At the base of the hierarchy is a small set of 'root keys' or 'hives', for example `HKEY_CURRENT_USER` for settings pertaining to the current logged-on user, and `HKEY_LOCAL_MACHINE` for settings specific to the computer. Beyond them, each key can have any number of 'values' as well as sub-keys, with each value having a name and some data. Values are typed to an extent, with the Registry itself defining three data types in particular — string, integer and 'binary', which basically means 'other'. As a convenience, `TRegistry` superimposes support for `Double` and `TDateTime`, both of which map to the 'binary' API type under the bonnet.

Different parts of the Registry can have different security levels. By default, an application running without administrator rights will only be able to write to the `HKEY_CURRENT_USER` hive, though even there individual keys can in principle have their own access levels. When manipulating the Registry programmatically, it is therefore important not to request more rights than are actually needed, for example requesting general read/write access when you only intend to read data.

Using TRegistry

Given this background, you normally use `TRegistry` by instantiating it, 'opening' or 'creating' a key, before reading or writing one or more values and finally freeing it. Once a key is opened, sub-keys and values can also be enumerated, keys and values renamed or deleted, and keys moved, saved and restored.

By default, `TRegistry` assumes you wish to read or write to the `HKEY_CURRENT_USER` tree. To work with another one (e.g. `HKEY_LOCAL_MACHINE`), set the `RootKey` property immediately after calling `TRegistry.Create`. Once `RootKey` is set, call either `OpenKey`, `OpenKeyReadOnly` or `CreateKey`, each of which take a Registry path (e.g. `'\Software\Hotsoft\Super App'`). `OpenKey` also has a second parameter to specify whether the requested key should be created if it doesn't already exist. On return, all three functions have a `Boolean` result indicating whether the key could actually be opened or created:

```
const
  ExtKey = '\Software\Classes\.myext';
var
  Registry: TRegistry;
begin
  Registry := TRegistry.Create;
  try
    //look to register globally first...
    Registry.RootKey := HKEY_LOCAL_MACHINE;
    if not Registry.OpenKey(ExtKey, True) then
    begin
      //failed, so try to register just for the current user...
      Registry.RootKey := HKEY_CURRENT_USER;
      if not Registry.OpenKey(ExtKey, True) then
        RaiseLastOSError;
    end;
    //...
```

As shown here, keys several layers down the hierarchy can be opened straight away. However, you can also go down in stages with successive calls so long as you *don't* prefix the path with a backslash. The following calls are therefore equivalent:

```
if Registry.OpenKeyReadOnly('\Parent\Child\Grandchild') then //...

if Registry.OpenKeyReadOnly('\Parent') and
   Registry.OpenKeyReadOnly('Child') and
   Registry.OpenKeyReadOnly('Grandchild') then //...
```

Calling `CloseKey` will reset the current key back to the root. This will also happen implicitly if you alter `RootKey` itself.

Enumerating keys and values

Once a key is open, you can call `GetKeyNames` to discover what sub-keys it has and `GetValueNames` for what values. Both methods fill a `TStrings` object that should have been created beforehand:

```
Registry.GetKeyNames(lsbKeys.Items);
Registry.GetValueNames(lsbValues.Items);
```

To just check for whether a given sub-key or value exists, call `KeyExists` or `ValueExists`. Aside from immediate children of the currently opened key, `KeyExists` also works with absolute paths (i.e., paths prefixed with a backslash) and for checking the existence of keys some way down the hierarchy (i.e., paths without a backslash and so relative to the open key):

```
if not Registry.OpenKeyReadOnly('\Control Panel') then
  RaiseLastOSError;
//both the following lines would normally output TRUE
WriteLn(Registry.KeyExists('Desktop\Colors'));
WriteLn(Registry.KeyExists('\Software\Microsoft\Windows'));
```

There is also a `HasSubKey` method for quickly determining whether the open key has any child keys, together with a `GetKeyInfo` function that fills in a `TRegKeyInfo` record:

```
type
  TRegKeyInfo = record
    NumSubKeys, MaxSubKeyLen, NumValues: Integer;
    MaxValueLen, MaxDataLen: Integer;
    FileTime: TFileTime;
  end;

function GetKeyInfo(var Value: TRegKeyInfo): Boolean;
```

The return value indicates whether information could be retrieved. Of the `TRegKeyInfo` fields, `FileTime` indicates when the key was last written to using the format of the underlying API. To convert to a `TDateTime`, call `FileTimeToLocalFileTime`, then `FileTimeToSystemTime`, and finally `SystemTimeToDateTime`:

```
function FileTimeToDateTime(const AFileTime: TFileTime): TDateTime;
var
  LocalTime: TFileTime;
  SysTime: TSystemTime;
begin
  if not FileTimeToLocalFileTime(AFileTime, LocalTime) or
     not FileTimeToSystemTime(LocalTime, SysTime) then
    RaiseLastOSError;
  Result := SystemTimeToDateTime(SysTime);
end;
```

As keys have `GetKeyInfo`, so values have `GetDataInfo`:

```
type
  TRegDataType = (rdUnknown, rdString, rdExpandString,
    rdInteger, rdBinary);

  TRegDataInfo = record
    RegData: TRegDataType;
    DataSize: Integer;
  end;

function GetDataInfo(const ValueName: string;
  var Value: TRegDataInfo): Boolean;
```

If you just want to learn a certain value's type, call `GetDataType`, or just its size, `GetDataSize`.

Reading values

Several methods are provided to read values: `ReadCurrency`, `ReadBinaryData`, `ReadBool`, `ReadDate`, `ReadDateTime`, `ReadFloat` (this reads a `Double`), `ReadInteger`, `ReadString`, `ReadTime`, and `GetDataAsString`. Most just take a single parameter, which specifies the value name, and should only be used to read data of the appropriate type — if you attempt to use `ReadInteger` on a value typed to a string (`REG_SZ`) rather than an integer (`REG_DWORD`), for instance, then an `ERegistryException` will be raised. Nonetheless, the `GetDataAsString` method does allow reading a string regardless of the actual type.

Another partial exception to the strong typing is `ReadBinaryData`, since this can be used to read either binary values or values with a type that `TRegistry` doesn't understand. In use, it takes an untyped buffer and the buffer size in bytes, similar to `TStream.ReadBuffer`. To read a binary value into a `TBytes` array, you might therefore use code like this:

```
function ReadBinaryBytesFromRegistry(Registry: TRegistry;
  const ValueName: string): TBytes;
begin
  SetLength(Result, Registry.GetDataSize(ValueName));
  if Result <> nil then
    Registry.ReadBinaryData(ValueName, Result[0], Length(Result))
```

```
end;
```

A helper routine for reading into a stream could then be layered on top:

```
procedure ReadBinaryStreamFromRegistry(Registry: TRegistry;
  const ValueName: string; Dest: TStream);
var
  Bytes: TBytes;
begin
  Bytes := ReadBinaryBytesFromRegistry(Registry, ValueName);
  if Bytes <> nil then Dest.WriteBuffer(Bytes[0], Length(Bytes));
end;
```

Data types

If you look back at the declaration of `TRegDataType` above, you will find its contents differ from what you might expect given the range of `ReadXXX` methods. This is primarily because `TRegistry` superimposes some types of its own beyond those natively supported by the Registry itself. In particular, it adds support for `Boolean`, date/time and floating point types. From the point of view of the operating system though, 'Boolean' values are really integers (DWORDs) that Delphi deems `False` if `0` and `True` otherwise, and the rest being 'binary' values.

Given that, the various elements of `TRegDataType` directly map to native Registry types: `rdString` maps to `REG_SZ`, `rdExpandString` to `REG_EXPAND_SZ`, `rdInteger` to `REG_DWORD`, and `rdBinary` to `REG_BINARY`. If a Registry value has a type `TRegistry` doesn't understand, then its type is reported as `rdUnknown`.

One slightly quirky type is `rdExpandString`/`REG_EXPAND_SZ`. This is for strings that contain unexpanded environment variables, i.e. sub-strings of the form `%VARNAME%`, such as `%SYSTEMROOT%` and `%USERPROFILE%`. In practice, `rdExpandString` is no different to `rdString` beyond giving a hint to the reader that the string may contain environment variables. This is why there is no `ReadExpandString` — instead, just use `ReadString`.

In order to actually expand anything, you must call the `ExpandEnvironmentStrings` API function, as declared in `Winapi.Windows`:

```
function ExpandEnvironmentVariables(const S: string): string;
var
  Len: DWORD;
begin
  Len := ExpandEnvironmentStrings(PChar(S), nil, 0);
  SetLength(Result, Len);
  ExpandEnvironmentStrings(PChar(S), PChar(Result), Len + 1);
end;

const
  P = '%USERPROFILE%\Stuff';
var
  S: string;
begin
  S := ExpandEnvironmentVariables(P); //C:\Users\CCR\Stuff
```

Dealing with Registry types not understood by TRegistry

As of Windows 7, RegEdit allows creating values with two types `TRegistry` does not understand: `REG_QWORD`, for 64 bit integers, and `REG_MULTI_SZ`, for 'arrays' of null-terminated strings.

In the case of these and indeed any other type not surfaced in `TRegDataType`, you can nevertheless still use `ReadBinaryData` to retrieve saved values in their 'raw' form. The following shows this in the case of a `REG_QWORD` value:

```
var
  Registry: TRegistry;
  Value: Int64;
begin
  Registry := TRegistry.Create;
  try
    if Registry.OpenKey('Software\Widget Corp\SuperApp', false) then
    begin
      if Registry.ReadBinaryData('Int64Test', Value,
         SizeOf(Value)) = SizeOf(Value) then
        WriteLn(Value)
      else
        WriteLn('Could not read Int64 value');
    end;
  finally
    Registry.Free;
  end;
```

```
end;
```

Reading `REG_MULTI_SZ` data is a bit more complicated, though only because the `REG_MULTI_SZ` format is itself more involved than a simple 64 bit integer:

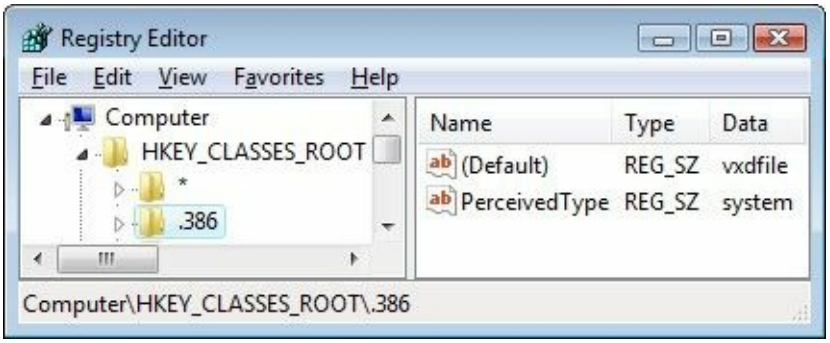
```
procedure ReadStringsFromRegistry(Registry: TRegistry;
  const ValueName: string; Strings: TStrings);
var
  Chars: array of Char;
  Len: Integer;
  SeekPtr: PChar;
  S: string;
begin
  Strings.BeginUpdate;
  try
    Strings.Clear;
    Len := Registry.GetDataSize(ValueName) div SizeOf(Char);
    if Len < 2 then Exit;
    //add two nulls so will always have valid data
    SetLength(Chars, Len + 2);
    Chars[Len] := #0;
    Chars[Len + 1] := #0;
    Registry.ReadBinaryData(ValueName, Chars[0], Len * SizeOf(Char));
    //scan through
    SeekPtr := @Chars[0];
    repeat
      S := SeekPtr;
      if Length(S) = 0 then Break; //got the double null
      Strings.Add(S);
      Inc(SeekPtr, Length(S) + 1);
    until False;
  finally
    Strings.EndUpdate;
  end;
end;
```

The routine works like this (if you aren't familiar with the `PChar` type, you may find it helpful to read the discussion in chapter 5 first):

- Each item of the 'array' is a null-terminated C-style string, with the end of the array denoted by a double null. For example, if `ReadBinaryData` returns a buffer containing `'Hello'#0'World'#0#0`, then the items of the 'array' are `'Hello'` and `'World'`.
- When we assign a `PChar` to a `string`, the characters up until the first null are copied over.
- Once done, the string is added to the list, the `PChar` pointer incremented past the null, and the assignment made again. If the resulting string is empty, then the double null and so the end of the list must have been reached. Otherwise, add the string to the list, seek past the latest null terminator, and test again.

'Default' values

If you browse the Registry using the Windows Registry Editor (RegEdit), you will find every single key having a 'default' value that is typed to `REG_SZ`. This oddity is due to historical reasons (originally, the Registry did not support multiple values per key). To read one, just call ReadString and pass an empty string. For example, if a key's values look like this in RegEdit:



Then the following will output `vxdfile` and `system`:

```
Registry.RootKey := HKEY_CLASSES_ROOT;
if Registry.OpenKeyReadOnly('\.386') then
begin
  WriteLn(Registry.ReadString(''));
```

```
  WriteLn(Registry.ReadInteger('PerceivedType'));
end;
```

Don't mix up `ReadString('')` with `ReadString('(Default)')`, since in the second case, `TRegistry` will look for a value whose name is actually `(Default)`!

Writing values

For each `ReadXXX` method there is a corresponding `WriteXXX` one too: `WriteCurrency`, `WriteBinaryData`, `WriteBool`, `WriteDate`, `WriteDateTime`, `WriteFloat`, `WriteInteger`, `WriteString`, `WriteExpandString`, `WriteTime`. Be warned that the normal case is for an `ERegistryException` to be raised if the value already exists but has a different type to the one you now wish to write.

When writing values with types `TRegistry` doesn't support natively, the Windows API (specifically the `RegSetValueEx` function) must be used directly. Even so, `TRegistry` can still be used up until the point of actually writing the value — after opening or creating a key, the operating system handle to it can be retrieved via the `CurrentKey` property. Support for writing `REG_QWORD` and `REG_MULTI_SZ` values, which we met a short while ago, may therefore be coded as thus:

```
uses
  Winapi.Windows, System.Classes, System.Win.Registry, System.RTLConsts;

const
  REG_QWORD = 11;

procedure WriteDataToRegistry(Registry: TRegistry;
  const ValueName: string; DataType: DWORD; const Data;
  DataLen: Integer);
begin
  { On failure, raise same exception TRegistry itself would raise }
  if RegSetValueEx(Registry.CurrentKey, PChar(ValueName), 0,
    DataType, @Data, DataLen) <> ERROR_SUCCESS then
    raise ERegistryException.CreateResFmt(@SRegSetDataFailed,
      [ValueName]);
end;

procedure WriteInt64ToRegistry(Registry: TRegistry;
  const ValueName: string; const Data: Int64);
begin
  WriteDataToRegistry(Registry, ValueName, REG_QWORD, Data,
    SizeOf(Data));
end;

procedure WriteStringsToRegistry(Registry: TRegistry;
  const ValueName: string; Strings: TStrings);
var
  Len: Integer;
  S, ToSave: string;
begin
  Len := 1;
  for S in Strings do
    Inc(Len, Length(S) + 1);
  if Len = 1 then
    ToSave := #0#0
  else
  begin
    SetLength(ToSave, Len);
    ToSave[Len] := #0;
    Len := 1;
    for S in Strings do
    begin
      MoveChars(S[1], ToSave[Len], Length(S) + 1); //1 for null
      Inc(Len, Length(S) + 1);
    end;
  end;
  WriteDataToRegistry(Registry, ValueName, REG_MULTI_SZ,
    ToSave[1], Length(ToSave) * SizeOf(Char));
end;
```

Moving and removing keys and values

Remove a key and all its sub-keys by calling `DeleteKey`, or just a specific value with `DeleteValue`:

```
Registry.DeleteKey('\Software\Widget Corp\SuperApp');
if Registry.OpenKey('\Software\Embarcadero\BDS\9.0') then
  Registry.DeleteValue('RegCompany');
```

`MoveKey`, as its name suggests, moves keys (including sub-keys) from one location to another. The following duplicates the `\Software\Widget Corp\SuperApp\1.0` structure to `\Software\Widget Corp\SuperApp\2.0`:

```
Registry.MoveKey('\Software\Widget Corp\SuperApp\1.0',
  '\Software\Widget Corp\SuperApp\2.0');
```

Be warned that `MoveKey` will silently fail if either the target key already exists or the source key doesn't exist. Similarly, a `RenameValue` method will silently fail if the target value already exists, or the old value doesn't exist.

Access levels

Whenever you use the `Open` method, `TRegistry` has to request a specific access level from the underlying API. By default, `KEY_ALL_ACCESS` is requested, which will lead to an exception being raised if your application attempts to open a key that it doesn't have the privileges to edit.

To fix this, you can pass an appropriate access level constant (typically `KEY_READ`) to the constructor, or set the `Access` property to it before `Open` is called:

```
uses Winapi.Windows, System.Win.Registry;

const
  Path = 'Software\Microsoft\Windows\CurrentVersion\Policies';
var
  Registry: TRegistry;
begin
  Registry := TRegistry.Create(KEY_READ);
  try
    if Registry.OpenKey(Path, False) then
```

Alternatively, you can call `OpenKeyReadOnly`. Helpfully, this will make several attempts, and sets the `Access` property to the successful request: first it tries requesting `KEY_READ`, then `KEY_READ` minus `KEY_NOTIFY` (alias `STANDARD_RIGHTS_READ or KEY_QUERY_VALUE or KEY_ENUMERATE_SUB_KEYS`), and finally just `KEY_QUERY_VALUE`. Consult MSDN or the Platform SDK for details concerning the meaning of these values.

Another reason for requesting a particular access level is to avoid the Registry's normal redirection behaviour: on 64 bit versions of Windows, certain system-defined keys and key hierarchies have different versions ('views') for 32 bit and 64 bit applications. Using `KEY_WOW64_32KEY` (`or`'ed if required) forces working with the 32 bit 'view', and `KEY_WOW64_64KEY` forces working with the 64 bit one:

```
uses
  System.SysUtils, System.Win.Registry, Winapi.Windows;

const
  Path = '\SOFTWARE\Microsoft\Windows\CurrentVersion';
  ValueName = 'ProgramFilesDir';
var
  Registry: TRegistry;
begin
  WriteLn('Looking under ', Path, ' for ', ValueName, '...');
  Registry := TRegistry.Create;
  try
    Registry.RootKey := HKEY_LOCAL_MACHINE;
    //force reading the 32 bit key
    Registry.Access := KEY_READ or KEY_WOW64_32KEY;
    if not Registry.OpenKey(Path, False) then
      RaiseLastOSError;
    WriteLn('32 bit value: ', Registry.ReadString(ValueName));
    //force reading the 64 bit key and 're-open' it
    Registry.Access := KEY_READ or KEY_WOW64_64KEY;
    if not Registry.OpenKey(Path, False) then
      RaiseLastOSError;
    WriteLn('64 bit value: ', Registry.ReadString(ValueName));
  finally
    Registry.Free;
  end;
  ReadLn;
end.
```

On my 64 bit Windows 7 computer, the first value retrieved is `C:\Program Files (x86)` and the second `C:\Program Files`.

One small 'gotcha' with `KEY_WOW64_32KEY` and `KEY_WOW64_64KEY` is that the `GetKeyNames` method is unaffected by either. As this limitation is inherited from the underlying API function (`RegEnumKeyEx`), it can't be helped. On the other hand, `TRegistry` itself will respect any redirection flag you set: in particular, when `OpenKeyReadOnly` succeeds, `KEY_WOW64_32KEY` or

`KEY_WOW64_64KEY` will be included in the new value for the `Access` property if either were included in it before.

INI-style configuration files (System.IniFiles)

The `System.IniFiles` unit provides a basic framework for simple configuration files. As its name suggests, it works in terms of the Windows INI format, i.e. text files that look like this:

```
[SectionName]
KeyName=Value
KeyName2=Value

[SectionName2]
KeyName=Value
```

In principle the framework can be given other sorts of backing store though, and further units in fact map it onto XML files and the Windows Registry.

TCustomIniFile

The core classes of `System.IniFiles` are a semi-abstract base class, `TCustomIniFile`, together with two concrete descendants, `TIniFile` and `TMemIniFile`, which map it to actual INI files.

The parent class exposes a property, `FileName`, which gets set in the constructor. While this will indeed contain a file name in the case of `TIniFile` and `TMemIniFile`, it may not for other descendants. `TCustomIniFile` also exposes the following members:

- Methods to test for and remove individual sections or keys (`SectionExists`, `EraseSection`, `ValueExists` [sic.] and `DeleteKey`).
- Various `ReadXXX` and `WriteXXX` methods for `Boolean`, `Double`, `Integer`, `string` `TDateTime`, `TDate`, `TTime` and `TStream` values. Most of the `ReadXXX` functions take a default value that will be returned when the key doesn't exist. The exception is `ReadBinaryStream`, which will return 0 since it ordinarily returns the number of bytes written to the destination stream. As implemented by `TCustomMemIniFile`, `WriteBinaryStream` encodes stream data as a string of hexadecimal characters, however descendant classes can override this. Similarly, the default behaviour for numerical or date/time values is to use their string representation, however this may also be overridden by sub-classes.
- A method (`UpdateFile`) for flushing changes to disk, if applicable. Not all `TCustomIniFile` descendants require this to be called (`TMemIniFile` does, but `TIniFile` and `TRegistryIniFile` don't).
- A few methods for reading things *en bloc*: `ReadSections` to load all section names into an existing `TStrings` instance, `ReadSection` (no 's') to load all key names for a given section into one, `ReadSectionValues` to load all keys for a given section using the format `Name=Value`, and `ReadSubSections` to load all 'sub-section' names for a given section. In the case of `ReadSectionValues`, use the `TStrings`' `Names`, `Values` and `ValueFromIndex` properties to read and manipulate the returned data. Be warned that support for sub-sections is rudimentary at best — by default, `ReadSubSections` uses backslashes to delimit sub-sections amongst ordinary section names.

To see these methods in action, imagine an INI file has the following contents, and is now being read by either `TIniFile` or `TMemIniFile`:

```
[MainForm]
Left=20
Top=100

[LastFile]
Path=C:\Users\CCR\Documents\Notes.txt

[MainForm\Toolbars\Standard]
New=1
Open=1
Save=0

[MainForm\Toolbars\Formatting]
Bold=1
Italic=0
```

(a) `ReadInteger('MainForm', 'Left', 0)` will return `20`.

(b) `ReadBool('MainForm\Toolbars\Formatting', 'Bold', False)` will return `True`.

(c) `ReadString('MainForm', 'Top', '')` will return `'100'`, notwithstanding the fact we appear to have numerical data.

(d) `ReadInteger('MainForm', 'Width', 440)` will return `440`, i.e. the specified default value.

(e) `EraseSection('MainForm')` will remove *just* the items under `[MainForm]` — those under the two 'sub-sections' will remain untouched. This behaviour cannot be relied upon for `TCustomIniFile` descendants generally however.

(f) Calling `ReadSections(Memo1.Lines)` will put the four strings `'MainForm'`, `'LastFile'`, `'MainForm\Toolbars\Standard'` and `'MainForm\Toolbars\Formatting'` into the memo control.

(g) Calling `ReadSubSections('MainForm', Strings)` will return nothing.

(h) However, `ReadSubSections('MainForm\Toolbars', Strings)` will return `'Standard'` and `'Formatting'`.

(i) `ReadSubSections('MainForm', Strings, True)` will return `'Toolbars\Standard'` and `'Toolbars\Formatting'`. The third parameter of `ReadSubSections` specifies whether to recurse or not, i.e. whether to return sub-sections of the immediate sub-sections and so on. Since no `[MainForm\Toolbars]` section is defined, *not* recursing like in example (g) leads to nothing being returned.

(j) `ReadSectionValues('MainForm', ListBox1.Items)` will put the strings `'Left=20'` and `'Top=100'` into list box. Subsequently, `ListBox1.Items.Names[1]` will return `'Top'`, and both `ListBox1.Items.Values['Left']` and `ListBox1.Items.ValueFromIndex[0]` will return `'20'`.

TIniFile vs. TMemIniFile

At the Windows API level, the 'standard' interface for reading and writing arbitrary INI files is constituted by the `GetPrivateProfileString` and `WritePrivateProfileString` functions. When targeting Windows, Delphi wraps these with the `TIniFile` class. However, alongside it is an independent implementation, `TMemIniFile`. This you may prefer for a couple of reasons: firstly, in being all Delphi code, it can be debugged into if necessary; and secondly, it supports Unicode (and in particular UTF-8) data with little fuss where `WritePrivateProfileString` (and therefore, `TIniFile`) does not. (On the Mac, `TIniFile` is a simple sub-class of `TMemIniFile` that just calls `UpdateFile` in its destructor.)

Whether you use `TIniFile` or `TMemIniFile`, you need to specify an appropriate path to the constructor. Unless writing a so-called 'portable' application that runs from a USB stick or the like, this will usually mean somewhere off of the `Application Data` (`AppData`) folder on Windows, and `~/Library` on OS X:

```
uses System.SysUtils, System.IniFiles;

function GetSettingsPath: string;
const
  SubDir = 'Widget Corp SuperApp';
begin
{$IFDEF MSWINDOWS}
  Result := IncludePathDelimeter(GetHomePath) + SubDir;
{$ELSE}
  Result := '~/Library/' + SubDir;
{$ENDIF}
  ForceDirectories(Result);
  Result := Result + PathDelim;
end;

var
  IniFile: TCustomIniFile;
begin
  IniFile := TMemIniFile.Create(GetSettingsPath + 'Settings.ini');
```

In use, `TMemIniFile` is little different to `TIniFile`, since you work with the interface defined by `TCustomIniFile`. However, there are a few differences.

The first is that you can specify a default text encoding when you instantiate `TMemIniFile`. For your own configuration files you will usually want to pass `TEncoding.UTF8` so that any strings with 'exotic' characters won't get garbled:

```
IniFile := TMemIniFile.Create(SomeFileName, TEncoding.UTF8);
```

If you don't specify an encoding and target Windows, then the active legacy 'Ansi' codepage will be assumed, though UTF-8 is the default when targeting OS X.

A second difference is that `UpdateFile` *must* be used to flush writes to disk — in contrast, this is just an optional call in the case of `TIniFile`. Relatedly, when you first construct a `TMemIniFile`, it loads all sections and keys into memory, only reloading them if you call the special `Rename` method and pass `True` for the second parameter:

```
MemIniFile.Rename(MemIniFile.FileName, True)
```

Pass a different file name and `False`, and the object will just save to the new file next time `UpdateFile` is called. Pass a different file name and `True`, and it will also reload itself from the new file.

The fact `TMemIniFile` loads everything up front can cause issues if multiple instances of your application are run at the same time. This is because, if you create a `TMemIniFile` object at start up, make changes to it as desired while the program is running, then call `UpdateFile` before freeing it on closing, then a race condition will exist between the different instances of the application as to whose changes will 'stick'.

Whether you care about this is up to you — I have come across large Microsoft programs that exhibit the problem, for example! Nonetheless, one way round it is to avoid keeping a 'live' `TMemIniFile` instance in the first place, and instead create and free one every time configuration settings are read or changed. Another possibility is to send some sort of signal to reload when the `TMemIniFile` of one instance is saved. In the case of a VCL application, this might take the form of a custom window message, in harness with the `BroadcastSystemMessage` Windows API function:

```
type
  TfrmMain = class(TForm)
  //...
    procedure FormCreate(Sender: TObject);
  strict private
    FIniFile: TMemIniFile;
    FReloadIniFileMsg: UINT;
    procedure FlushIniFileChanges;
  protected
    procedure WndProc(var Message: TMessage); override;
  end;

//...

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  FReloadIniFileMsg := RegisterWindowMessage('SuperAppReloadIniFile');
end;

procedure TfrmMain.FlushIniFileChanges;
var
  Recipients: DWORD;
begin
  FIniFile.UpdateFile;
  Recipients := BSM_APPLICATIONS;
  BroadcastSystemMessage(BSF_FORCEIFHUNG or BSF_IGNORECURRENTTASK or
    BSF_POSTMESSAGE, @Recipients, FReloadIniFileMsg, 0, 0);
end;

procedure TfrmMain.WndProc(var Message: TMessage);
begin
  if Message.Msg = FReloadIniFileMsg then
    FIniFile.Rename(FIniFile.FileName, True);
  inherited;
end;
```

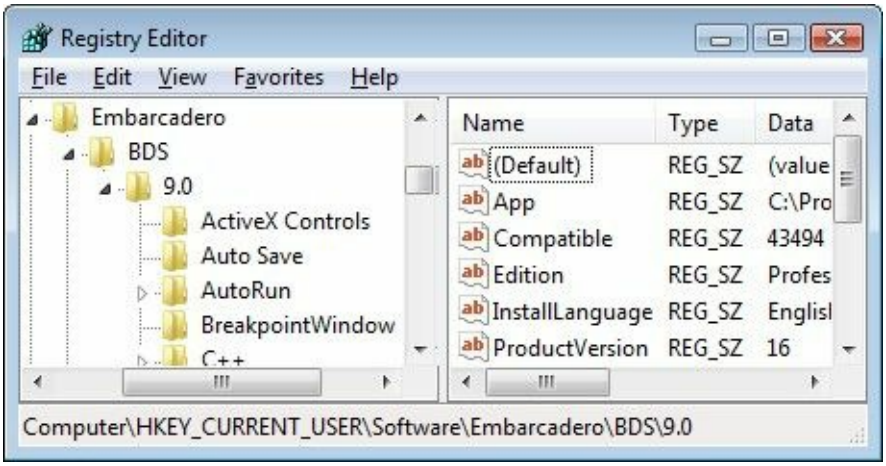
TCustomIniFile descendants that don't read and write to INI files

In the box you can find a couple `TCustomIniFile` descendants that don't read and write to actual INI files: `TRegistryIniFile` (declared in `System.Win.Registry`), which interfaces with the Windows Registry, and `TXmlIniFile` (declared in `Xml.XmlIniFile`), which works on a loaded XML file. You shouldn't consider the classes involved as general Registry and XML parsing tools — rather, they exist to map the `TCustomIniFile` interface onto those other media.

On construction, you pass `TRegistryIniFile` not a file name, but a Registry path and (optionally) Registry access level, the latter using one or more constants declared in `Winapi.Windows`, e.g. `KEY_READ` (see MSDN or the Platform SDK for a full list). For example, the following creates a `TRegistryIniFile` instance that points to the user settings for Delphi or RAD Studio XE2, and requests read access only:

```
IniFile := TRegistryIniFile.Create(
  '\Software\Embarcadero\BDS\9.0', KEY_READ);
```

Since the Registry itself speaks in terms of 'keys' and 'values', there is a small clash of terminology. The mapping goes like this: the Registry keys immediately below the key whose path is given to `Create` become the 'sections', and their values the INI 'keys'. Thus, in the previous example, the 'sections' will be `'ActiveX Controls'`, `'Auto Save'`, `'AutoRun'` and so on, with the 'keys' for `'ActiveX Controls'` being `'PalettePage'` and `'UnitDirName'` on my machine:



The Windows Registry is organised into sub-trees, and by default, `TRegistryIniFile` reads and writes under `HKEY_CURRENT_USER`. To work with a different sub-tree, use the `RegIniFile` property to access the underlying `TRegistry` object, and change it from there (without special privileges it is unlikely `TRegistryIniFile` will be able to write to another root however):

```
IniFile := TRegistryIniFile.Create('', KEY_READ);
try
  IniFile.RegIniFile.RootKey := HKEY_CLASSES_ROOT;
  if not IniFile.RegIniFile.OpenKey('.doc', False) then
    raise Exception.Create('DOC extension not registered!');
```

Nonetheless, for your own settings, usual practice is to write to `HKEY_CURRENT_USER`, and within that, a path along the lines `\Software\Your Company Name\Your Application Name`:

```
IniFile := TRegistryIniFile.Create(
  '\Software\Widget Corp\SuperApp');
```

TRegistryIniFile write behaviour

`TRegistryIniFile` implements its inherited `WriteXXX` methods to use the corresponding Registry data type if one exists:

- `WriteBinaryStream` will use the `REG_BINARY` Registry data type.
- `WriteInteger` will use `REG_DWORD`, unless the key already exists and has a string type, in which case `REG_SZ` will be used. Similarly, `ReadInteger` will work if the value exists, is numeric in content, but has the `REG_SZ` type. This contrasts with the equivalent methods of `TRegistry`, which would raise an exception in both instances.
- `WriteString` will use `REG_SZ`.
- Since the Windows Registry does not have a floating point data type, `REG_BINARY` is used again for `WriteDate`, `WriteTime`, `WriteDateTime` and `WriteFloat`.

By its very nature, `TRegistryIniFile` is specific to Windows. As shipped, there is no equivalent class in the box for OS X. On the one hand this is not surprising, since OS X does not have a central registry like Windows. On the other though, OS X does have a 'preferences' API that would seem ideally suitable for wrapping in a `TCustomIniFile` shell.

You can find such a class in this book's accompanying source code, which can be considered an example of what it takes to write a fully-featured `TCustomIniFile` descendant. Just as `TRegistryIniFile` can only serve as a general purpose Registry reader and writer at a pinch, so my class isn't really designed to read and write from Preferences files you don't define yourself, though it can work with them. Nonetheless, it writes using the native types, which in its case means the appropriate 'property list' types — `CFNumberRef`, `CFDateRef`, etc.

TXmlIniFile

`TXmlIniFile` works with an independently-managed `IXmlDocument` instance: pass it an `IXmlNode` to treat as the root, and child nodes become the INI 'sections' and attributes used for the 'keys'.

For example, consider the following code:

```
uses Xml.XmlIntf, Xml.XmlDoc, Xml.XmlIniFile;

var
  Doc: IXMLDocument;
  IniFile: TXmlIniFile;
begin
```

```
  Doc := NewXMLDocument;
  Doc.Options := Doc.Options + [doNodeAutoIndent];
  Doc.DocumentElement := Doc.CreateNode('Sections');
  IniFile := TXmlIniFile.Create(Doc.DocumentElement);
  try
    IniFile.WriteInteger('ImportantNumbers', 'Emergency', 999);
    IniFile.WriteInteger('ImportantNumbers', 'Callback', 1471);
    IniFile.WriteString('LastUser', 'Name', 'Joe Bloggs');
    IniFile.WriteBool('LastUser', 'LoggedInOK', True);
  finally
    IniFile.Free;
  end;
  Doc.SaveToFile(ChangeFileExt(ParamStr(0), '.xml'));
end.
```

This will output an XML file that looks like this:

```
<?xml version="1.0"?>
<Sections>
  <ImportantNumbers>
    <ImportantNumbers Name="Emergency">999</ImportantNumbers>
    <ImportantNumbers Name="Callback">1471</ImportantNumbers>
  </ImportantNumbers>
  <LastUser>
    <LastUser Name="Name">Joe Bloggs</LastUser>
    <LastUser Name="LoggedInOK">True</LastUser>
  </LastUser>
</Sections>
```

In this case, the node used as the 'INI' root is the document element node itself, i.e. the root node of the whole XML structure. However, in principle the 'INI' root could be several elements deep.

Partly because of this, calling `UpdateFile` against a `TXmlFile` object does nothing — instead, you need to the `SaveToFile` or `SaveToStream` methods of the `IXMLDocument` instance you began with. To change this behaviour and make `TXmlFile` truly standalone, a small descendant class needs to be written:

```
type
  TStandaloneXmlIniFile = class(TXmlIniFile)
  strict private
    FDoc: IXMLDocument;
  public
    constructor Create(const AFileName: string);
    procedure UpdateFile; override;
  end;

constructor TStandaloneXmlIniFile.Create(const AFileName: string);
begin
  if FileExists(AFileName) then
    FDoc := LoadXMLDocument(AFileName)
  else
  begin
    FDoc := NewXMLDocument;
    FDoc.Options := FDoc.Options + [doNodeAutoIndent];
  end;
  if FDoc.DocumentElement = nil then
    FDoc.DocumentElement := FDoc.CreateNode('Sections');
  TCustomIniFile(Self).Create(AFileName); //set FileName prop
  inherited Create(FDoc.DocumentElement);
end;

procedure TStandaloneXmlIniFile.UpdateFile;
begin
  FDoc.SaveToFile(FileName);
end;
```

This loses the flexibility of choosing which node to parent to however.